Apiary Hands-On Lab

You will have been assigned a particular service to work on for the following exercises, throughout this document it will be referred to as `your service`.

The following exercises are designed to provide a basic understanding of how Apiary and the API Design-First process work here at IPIM-IP.

If you encounter any issues during the following exercises please ask one of the on-site support team for help.

Table of Contents

- Apiary Hands-On Lab
  - Table of Contents
  - Setup
    - Required Tools
    - Useful URLs
  - Section 1 - Introduction to Apiary
    - Documentation Tab
      - Trying a Resource
    - Exercise 1: Mock Server
      - Exercise 1 - Required Tools
      - Exercise 1 - Task
      - Exercise 1 - Success Criteria
  - Section 2 - The Editor Tab
    - Document Format and Host
    - Document Information
    - Logical Groups
    - Specific Requests
      - Specific Request Parameters
        - Query Parameters
        - Path Parameters
        - Parameters
    - Exercise 2: Adding New Parameters
      - Exercise 2 - Required Tools
      - Exercise 2 - Task
      - Exercise 2 - Success Criteria
  - Section 3 - Data Structures
    - Simple Objects
    - Complex Objects
    - Exercise 3: Adding a Data Structure
      - Exercise 3 - Required Tools
      - Exercise 3 - Task
      - Exercise 3 - Success Criteria
  - Section 4 - Request and Response Bodies

        - Specific Request Bodies
        - Specific Response Bodies
        - Exercise 4.1: Modifying Request & Response Bodies
            - Exercise 4.1 - Required Tools
            - Exercise 4.1 - Task
            - Exercise 4.1 - Success Criteria
        - Exercise 4.2: Adding Request and Response Bodies
            - Exercise 4.2 - Required Tools
            - Exercise 4.2 - Task
            - Exercise 4.2 - Success Criteria
    - Section 5 - Tying Everything Together
        - Exercise 5: Adding New Endpoints
            - Exercise 5 - Required Tools
            - Exercise 5 - Task
            - Exercise 5 - Success Criteria

Setup

After logging in to your provided virtual machine you should see your service downloaded to the machine along with suitable development tools such as Postman, Visual Studio Code, Notepad++ etc.

Required Tools

For the following exercises, the following tools are required:

- Internet Browser (Google Chrome)

Useful URLs

- Apiary Lab (This Document)
- Apiary
- Apiary Documentation
- Microservices workbench

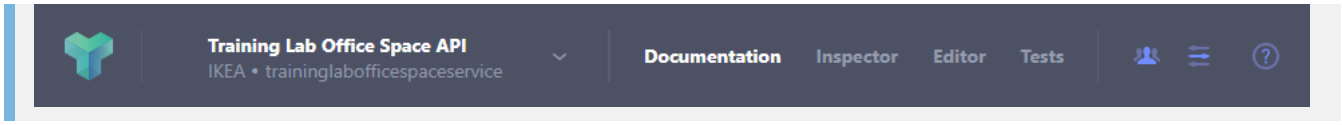
> *Note: Please **only use the below link for your service** as other people will be working on their services.*

- Bedroom Service
- Kitchen Service
- Living Room Service
- Storage Service
- Office Space Service

Section 1 - Introduction to Apiary

You should have received an invitation to work on your apiary service, please follow the instructions in that email to create your account if you do not already have one.

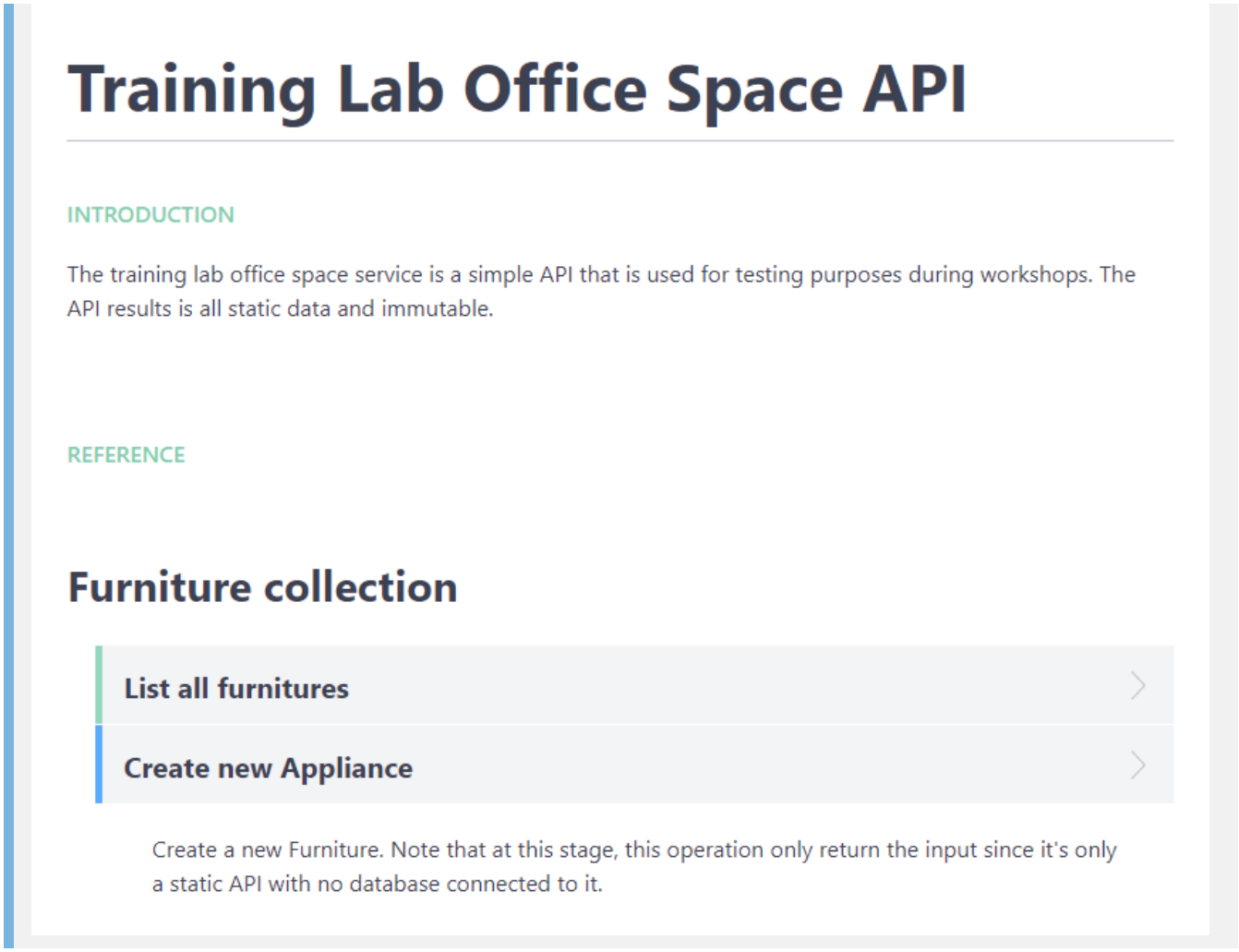
The Apiary application has a menu section located at the top of the page, the two key tabs are **Documentation** and **Editor**



Documentation Tab

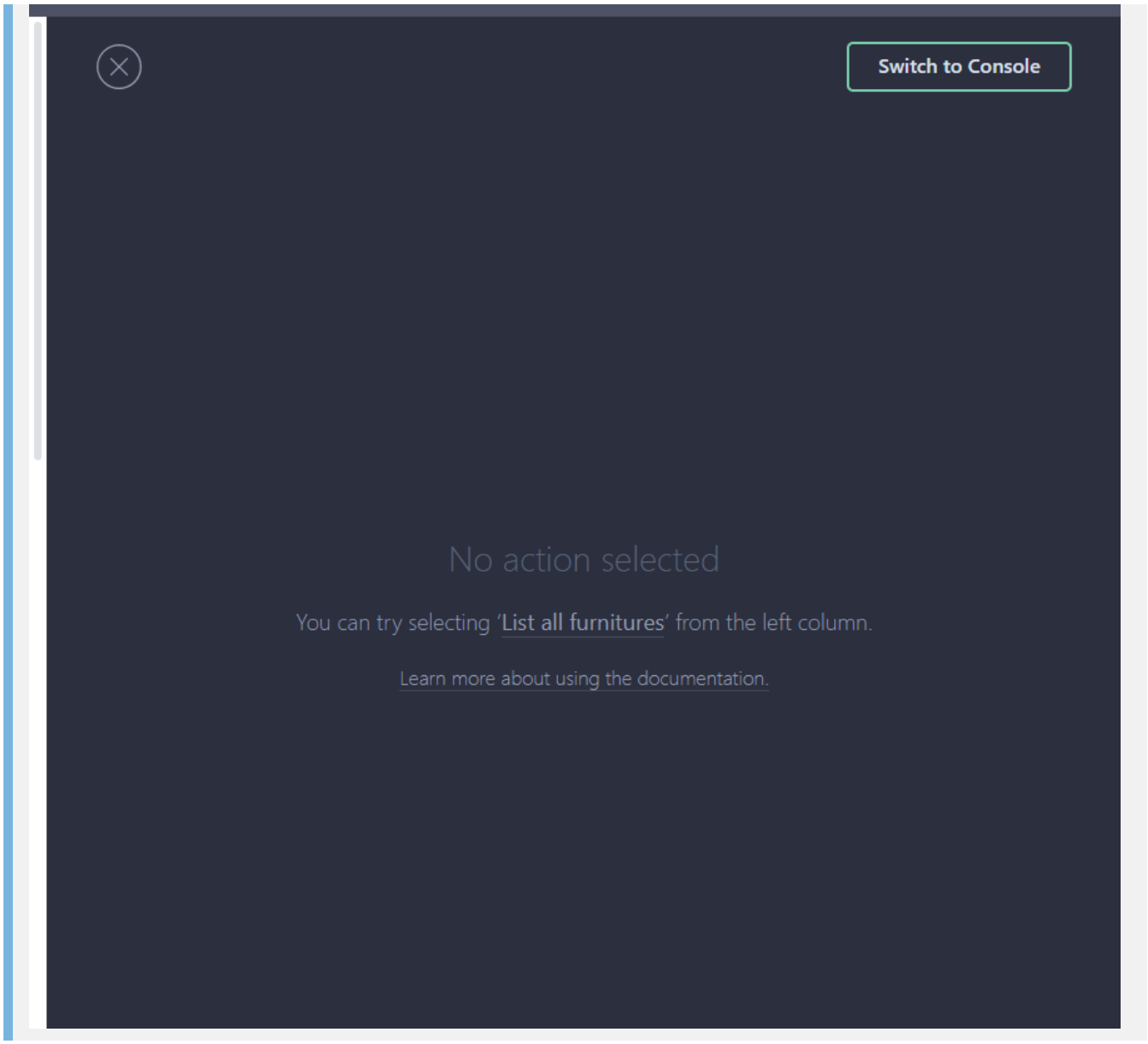
The documentation tab contains the rendered API Design documentation, this is the document that any potential API Consumer would look at in order to understand how your API works, due to this reason it is extremely important to ensure the documentation is accurate and up-to-date otherwise it can have a negative impact on whether your API is to be consumed in the future.

The document contains a high level introduction followed by one or logical groups, with each logical groups potentially containing multiple API resources.

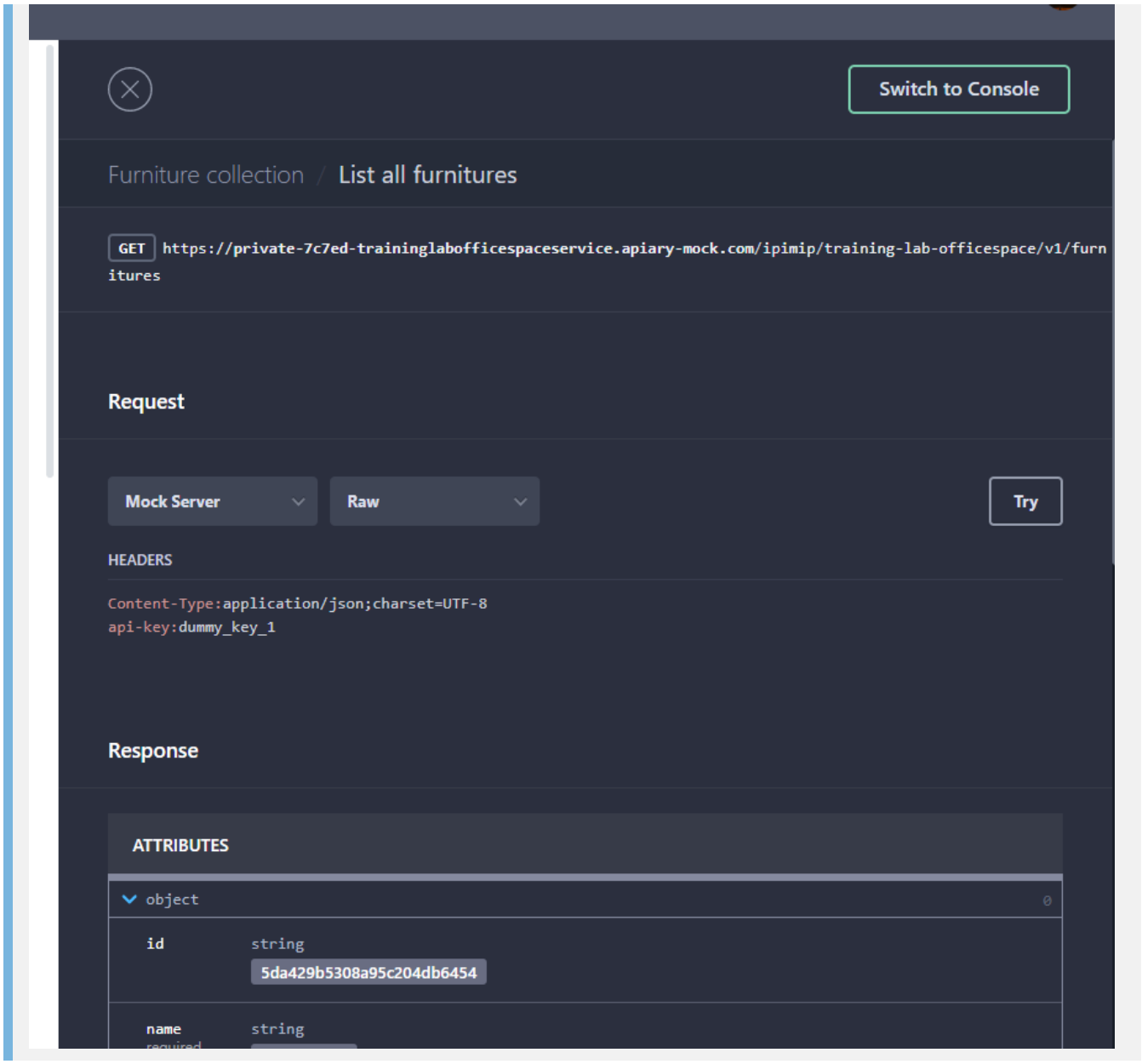


Trying a Resource

If you want to drill into a particular API resource, you can select one of the named resources in the document and it opens the right hand panel, often referred to as the Example area.

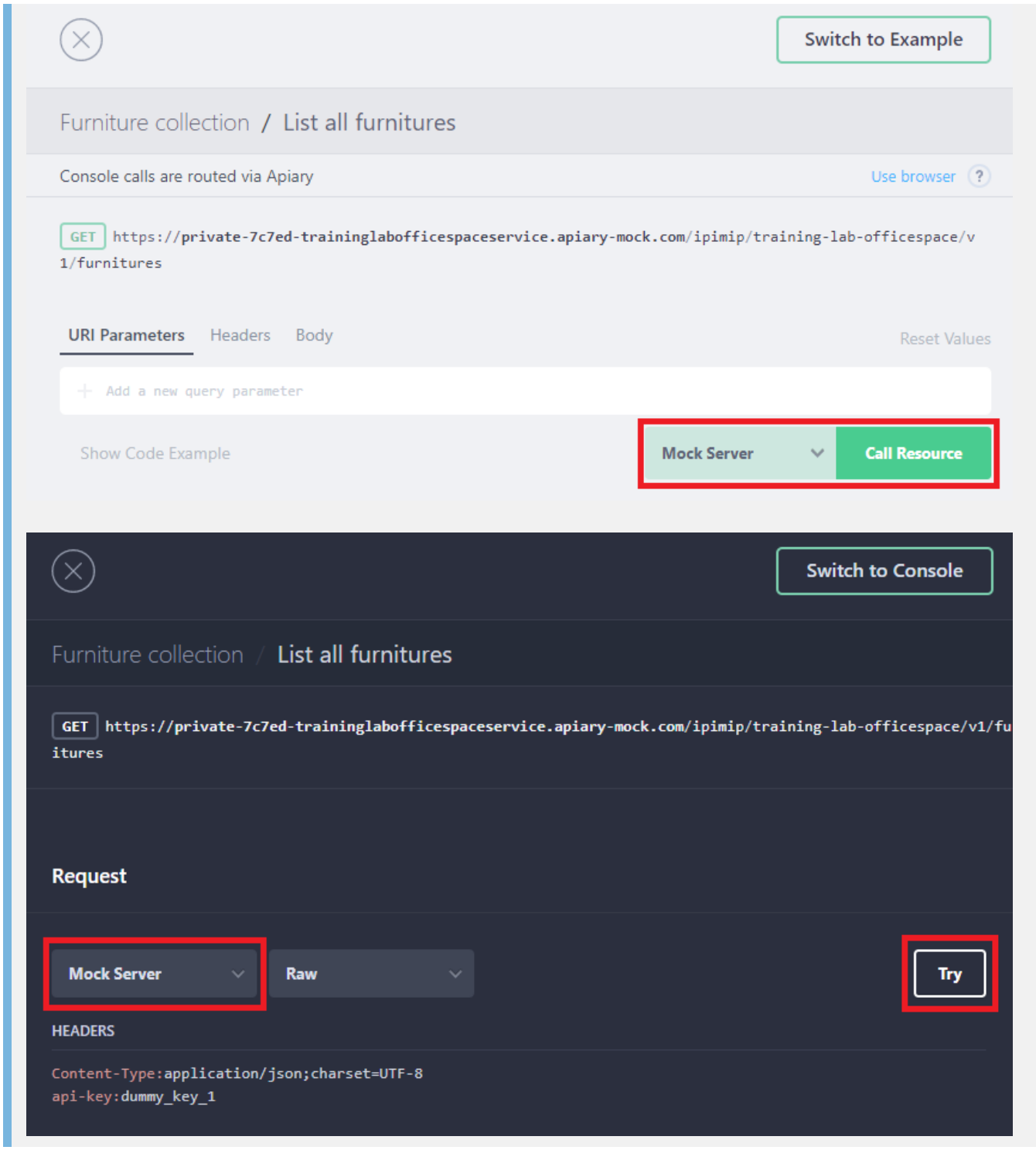


The example area allows a consumer to see the URI of the request, any applicable parameters (Query or Path) along with example request (if applicable) and response bodies.

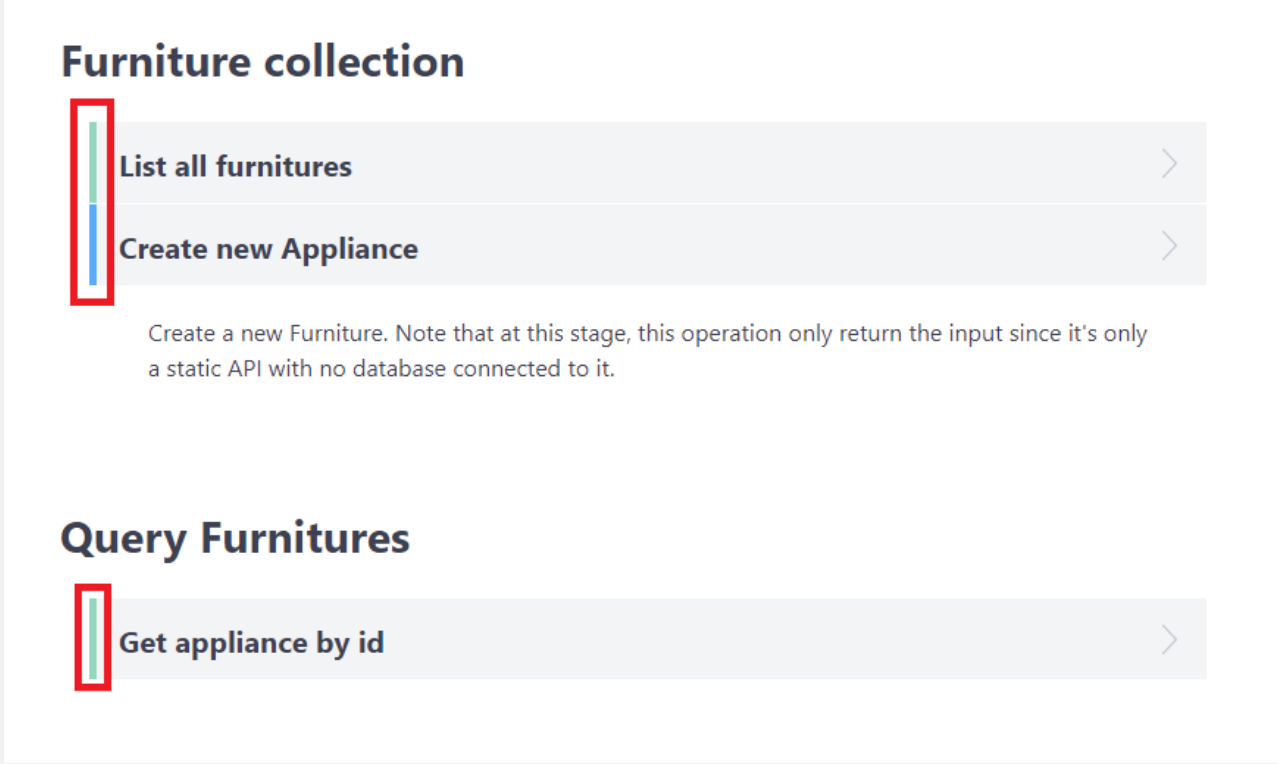


You are able to switch the panel on the right hand side from Example to Console, a key feature of this is the Mock Server setting, this feature allows you to test out the API design as though it was a real API, using your favourite API Client or the Apiary application you can call the end-point and see the example responses you have created be returned. This is one of the key features of the API Design first approach as it allows consumers to call the Mock Server in order to get a response payload that matches what the real service will return, using the Mock Server allows consumers to **not** be dependant on the development of the service as they are able to consume the API as soon as the design is completed.

In order to open the Mock Server you can click *Try* via the Example screen or *Call Resource* in the Console screen.



The remainder of the document in Apiary's documentation tab contains all of the resources your API exposes, as this is the consumer documentation it is important that all of the possible resources are documented.



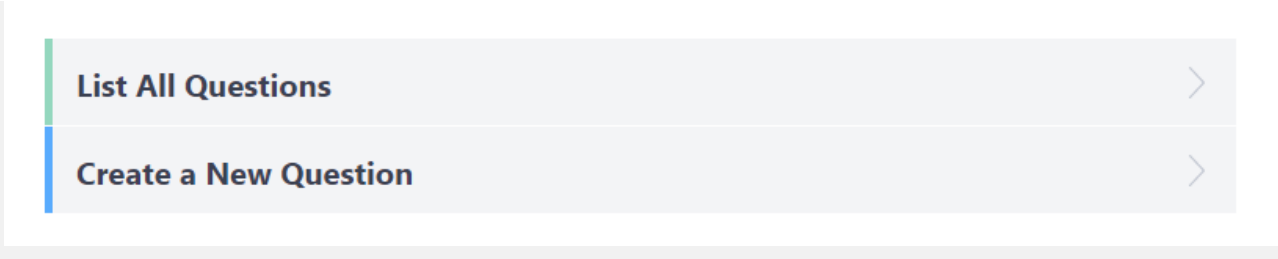
Note: The colour of the resource determines the type, Green = GET, Blue = POST, Grey = PUT, RED = DELETE and Brown = PATCH

Exercise 1: Mock Server

This exercise is designed to familiarise yourself with the Mock Server aspect of the Apiary application - As further exercises require you to edit the Apiary document for your service, please use the below Apiary document for this exercise:

- Example Poll API Document

The various resources in the Apiary document look like the below image:



Exercise 1 - Required Tools

- Internet Browser

Exercise 1 - Task

Within Apiary, on the documentation tab, try all of the available resources that the Polls API has exposed.

Pay attention to the request and response sections for the different HTTP Methods in the document.

> *Tip: Check the `Body` section when making a POST request*

Exercise 1 - Success Criteria

- [ ] Try the `List All Questions` resource
- [ ] Try the `Create a New Question` resource
- [ ] Familiar with the Apiary Documentation tab

Section 2 - The Editor Tab

The editor tab consists of two panels, the left panel contains the raw, un-processed API design document, within Apiary this can either be an API Blueprint or Swagger/OAS document, whilst the right panel is what you would see if you clicked on the documentation tab.

> *Note: For the exercises in this lab we will be using API Blueprint notation as in our experience we have found it easier to work with when you're new to API Design.*

Document Format and Host

On the left panel, the first thing you will see is the FORMAT and HOST properties, the FORMAT should be set to `1A` for API Blueprint whilst the HOST field should be set to the real-life API gateway URL including the API base path, see the example below:

```
FORMAT: 1A
HOST: https://dev.westeurope.api.hip.red.cdtapps.com/ipimip/training-lab-officespace
```

API Gateway URL

API Base Path

Document Information

With the Format and Host being set the next item you enter is the API Document title, this uses Markdown syntax which uses `#` characters to decide on the heading size, the largest heading is `#` whereas the smallest heading is `#####`, each API document should contain **one** `#` level heading or **two** `#` level headings if using the MSON syntax, these are usually the first and last items in the API document.

After the document title you are able to enter a description/overview of the API document if you wish to do so, this isn't mandatory but it usually sets the background picture for any API consumer looking over your documentation.

```
# Training Lab Office Space API

The training lab office space service is a simple API that is used for testing purposes during wor
The API results is all static data and immutable.
```

Logical Groups

After the document title and description has been set, you are now free to create logical groups of resources, a logical group of resources is where the URI remains the same for all operations, in the example below, the GET and POST methods are all executed on the same endpoint which therefore makes it logical to group them together.

When defining a logical group, you are also required to enter the URI endpoint that should be called for this group, the endpoint is denoted by being placed inside square brackets `[]`.

> *Note: A logical group can be defined using the `##` notation*
>
> ```
> ## Furniture collection [/v1/furnitures]
>
> Logical Group 1
>
> ## Query Furnitures [/v1/furnitures/{id}]
>
> Logical Group 2
> ```

Specific Requests

Within a logical group, you are able to define specific requests that can be called on the endpoint, for each request defined you must also specify the http method used to invoke the request by placing the method name in square brackets `[]`.

> *Note: One request can be defined using the `###` notation.*

After you have specified the name and method type of the request you can choose to enter a description for the specific resource.

```
### Create new furniture [POST]
Create a new Furniture. Note that at this stage, this operation only return the input since it's
```

You are required to specify additional information required in order to call the specific resource, the additional information is often but is not limited to:

- Parameters
- Request Bodies
- Response Bodies

Specific Request Parameters

Certain URIs require parameters to be included as part of the request url, there are two main types of parameters, Query and Path. Within Apiary it is possible to include Path and Query parameters in the same request if required.

Query Parameters

A `Query Parameter` is included at the end of the URL following a `?` character.

> /resource`?query-parameter-1=ABC&query-parameter-2=DEF`

The above URI contains two query parameters, named `query-parameter-1` and `query-parameter-2`, the values of these query parameters are `ABC` and `DEF` respectively.

```
## Query Furnitures [/v1/furnitures{?furniture%2Dcode,type}]
```

> The notation above shows how you can enter query parameters in the Apiary document, in this particular example the query parameters both contain a `-` character therefore the HTTP representation `%2D` is used to ensure it is rendered correctly on the right hand panel.
>
> ```
> GET https://private-7c7ed-traininglabofficespaceservice.apiary-mock.com/ipimip/training-lab-officespace/v1/furnitures?furniture-code=CODE123&type=TABLE
> ```

Path Parameters

A `Path Parameter` can be included at any point in the URI and is often used to provide context.

> /vehicle/`{vehicle-id}`/drivers/`{driver-id}`

The above URI contains two path parameters, named `vehicle-id` and `driver-id`, the values of the path parameters are set when you call the URI, the first parameter is used to retrieve a specific vehicle based on it's vehicle ID from a list of vehicles, the second parameter is used to retrieve a specific driver based on the `driver-id` from a list of drivers that are associated to one specific vehicle (The one identified by `vehicle-id`).

> ```
> ## Query Furnitures [/v1/furnitures/{id}]
> ```
>
> The notation above shows how you can enter path parameters in the Apiary document, in this particular example their is only one path parameter `{id}` at the end of the URI although it is possible to include multiple path parameters.
>
> ```
> GET https://private-7c7ed-traininglabofficespaceservice.apiary-mock.com/ipimip/training-lab-officespace/v1/furnitures/id
> ```

Parameters

For both query and path parameters, you have to include a specific parameter section inside the specific request, this can be achieved using the `+ Parameters` notation. Each parameter should be on it's own line and be indented by **four spaces** or **one TAB** character.

> *Note: The names of the parameters should match the ones defined in the URI*
>
> ```
> + Parameters
>  + furniture%2Dcode: CODE123 (string, required) - The furniture code
>  + type:             TABLE   (string, required) - The type of furniture
> ```
>
> **Parameters**
>
> | | | |
> |---|---|---|
> | **furniture-code** | The furniture code Example: `CODE123`. | String |
> | **type** | The type of furniture Example: `TABLE`. | String |

Exercise 2: Adding New Parameters

This exercise will introduce you to editing the Apiary document by adding new query parameters for the `Query` resource, by the end of this exercise you should be familiar with editing the Apiary document and also the parameters section.

> *Note: **Please use your services API Document for this and subsequent exercises - You will also need to switch to the Editor tab**:*

- Bedroom Service
- Kitchen Service
- Living Room Service
- Storage Service
- Office Space Service

> *Note: Refer back to the Specific Request Parameters Section for more information on Apiary parameters*

Exercise 2 - Required Tools

- Internet Browser

Exercise 2 - Task

Currently the `Query` resource allows you to search for a particular furniture item using the `Type` field, your task is to change the `Query` resource so that you are now able to query on the `Type` and `Name` fields.

> *Tip: Remember to add the new parameter in two places*

Exercise 2 - Success Criteria

- [ ] Add new query parameter `Name`
- [ ] Try the updated `Query` resource
- [ ] Familiar with the Apiary Editor tab
- [ ] Remove comments related to this exercise

Section 3 - Data Structures

The following exercises are using the MSON (Markdown Syntax for Object Notation) Syntax.

MSON is a plain-text, human readable way of documenting data structures and therefore is a great choice if you're new to API Design as it compliments your existing business knowledge without becoming too technical.

The data structures are usually placed at the end of the document in order to help a user understand the various schema.

Each object should contain various fields, each field should have a name and type as a minimum although it could contain a default value, sample value and description denoted using the following

syntax:

> `- fieldName: fieldDefaultValue/fieldSampleValue (type, [required/optional], [sample]) - fieldDescription`
>
> e.g. `- furnitureName: A long dark door (string, required) - The name of the furniture item`
>
> *Note: To escape certain field names or values in the document surround the value using the ` character i.e. `24-10-2019` escapes the `-` character*

Simple Objects

The primitive list of types for Apiary are:

- boolean - true/false
- string - Any string
- number - Any number

Each data structure you create can be declared as an object itself by adding `(object)` to the name of the data structure. This is discussed in further detail in the Complex Objects section.

> ```
> ## Furniture (object)
>
> - id: 5da429b5308a95c204db6454 (string, optional, sample) - The ID of the furniture item
> - name: KULLABERG (string, required, sample) - The name of the furniture item
> - type: Desk (string, required, sample) - The type of the furniture item
> - price: 1500.00 (number, required, sample) - The price of the furniture item
> ```

The above example shows a data structure called Furniture being created, this new structure contains four fields and each of the fields contains the relevant associated information.

> *Note: If the API defaults a value if none is provided for a specific field then **do not** include the `sample` type on the field and the value after the field name will be used as the default value as opposed to a sample value*

Complex Objects

Often the data structures that should be returned from an API are complex and could contain special fields like an array, Apiary currently supports the following structure types:

- array - List of items
- enum - Exclusive list of possible values
- object - A structure containing fields/members

For example, using the Furniture object above, it could be extended to include an Array type for one of it's fields.

```
## Furniture (object)

- id: 5da429b5308a95c204db6454 (string, optional, sample) - The ID of the
furniture item
- name: KULLABERG (string, required, sample) - The name of the furniture
item
- type: Desk (string, required, sample) - The type of the furniture item
- price: 1500.00 (number, required, sample) - The price of the furniture
item
- colour: red, green (array[string], optional, sample) - The colour(s) of
the furniture item
```

Another common use case is to have nested objects, an object residing inside another object, using the furniture example above, you can add a materials field that contains an array of Material objects.

```
## Furniture (object)

- id: 5da429b5308a95c204db6454 (string, optional, sample) - The ID of the
furniture item
- name: KULLABERG (string, required, sample) - The name of the furniture
item
- type: Desk (string, required, sample) - The type of the furniture item
- price: 1500.00 (number, required, sample) - The price of the furniture
item
- colour: red, green (array[string], optional, sample) - The colour(s) of
the furniture item
- materials (array[Material], required) - The materials used to make the
furniture item

## Material (object)
- id: 123456789 (number, required, sample) - The id of the material
component
- type: wood (string, required, sample) - The type of the material
- length: 100 (string, required, sample) - The length of the material in mm
```

Now the Material object has been created and the Furniture object uses this in it's materials field which contains an array of type `Material`, when rendered in the documentation this is how it would look.

ATTRIBUTES		
id	string	The ID of the furniture item
name required	string	The name of the furniture item
type required	string	The type of the furniture item
price required	number	The price of the furniture item
colour	array	
materials required	array	The materials used to make the furniture item
object		
id required	number	The id of the material component
type required	string	The type of the material
length required	string	The length of the material in mm

The final example of complex types is where a type inherits the fields from another type, for example if you have a set of fields that are often re-used across multiple objects then those fields can be placed into their own object and can be inherited.

```
## Furniture (AuditFields)
- id: 5da429b5308a95c204db6454 (string, optional, sample) - The ID of the
furniture item
- name: KULLABERG (string, required, sample) - The name of the furniture
item
- type: Desk (string, required, sample) - The type of the furniture item
- price: 1500.00 (number, required, sample) - The price of the furniture
item
- colour: red, green (array[string], optional, sample) - The colour(s) of
the furniture item

## AuditFields (object)
```

```
- creationDate: `29-09-2019` (string, required, sample) - The creation date
of the object
- createdBy: user1 (string, required, sample) - The user id who created the
object
- lastUpdatedDate: `14-10-2019` (string, required, sample) - The date the
object was last updated
- lastUpdatedBy: user2 (string, required, sample) - The user id who last
updated the object
```

Now you can see that the Furniture object inherits all of the fields on the AuditFields object, it's worth noting that any field inherited using the above approach will always appear at the start of the object, i.e. the AuditFields fields will be rendered before the Furniture fields.

```
{
  "creationDate": "29-09-2019",
  "createdBy": "user1",
  "lastUpdatedDate": "14-10-2019",
  "lastUpdatedBy": "user2",
  "id": "5da429b5308a95c204db6454",
  "name": "KULLABERG",
  "type": "Desk",
  "price": 1500,
```

More detailed documentation can be found on the MSON GitHub repository.

Exercise 3: Adding a Data Structure

This exercise will introduce you to the Data Structure sections of the Apiary document.

Exercise 3 - Required Tools

- Internet Browser

Exercise 3 - Task

Currently the Data Structures section of the document does not contain a Furniture object, you need to create a furniture object so it can be re-used across multiple resources.

The entire furniture object contains the fields and all fields are required:

- id
- name
- type
- price

Remember to add a sample value for each field.

> *Tip: The basic primitive types in Apiary are `string` and `number`*

Exercise 3 - Success Criteria

- [ ] New data structure named "Furniture"
- [ ] Appropriate data types chosen for the four fields
- [ ] Remove comments related to this exercise

Section 4 - Request and Response Bodies

Specific Request Bodies

In order to start the request section, you use the notation `+ Request`, any information on lines after this will be treated as part of the request body.

You are also able to specify the request content-type if the request body contains data, this is done by adding `(content-type)` to the end of the `+ Request` line.

> *For Example: `+ Request (application/json;charset=UTF-8)` means a Request Body with Content-Type `application/json;charset=UTF-8`*
>
> ```
> + Request (application/json;charset=UTF-8)
> ```

If the request body contains any data/attributes then you include these in the line underneath `+ Request`, for API blueprint to be rendered properly, the data/attributes should be indented by **four spaces** or **one TAB** character and is denoted using the `Attributes` field.

> ```
> + Request (application/json;charset=UTF-8)
>
> + Attributes (Furniture)
> ```

Additionally you are able to specify any additional headers to be used on the request, for IPIM-IP we often include as a minimum an *api-key* which is required to be passed for any API call, to add custom headers in the Apiary document they should be indented by **twelve spaces** or **three TAB** characters from the `+ Request` heading and the headers are denoted using the `Headers` field.

> ```
> + Request (application/json;charset=UTF-8)
>
> + Attributes (Furniture)
>
> + Headers
>
> api-key: dummy_key_1
> example_header: 1234
> ```

Specific Response Bodies

In order to start the response section, you use the notation `+ Response`, any information on lines after this will be treated as part of the response body. Along with declaring the start of the response section, you also have to specify the HTTP status code that the response should return.

You are also able to specify the response content-type if the response body contains data, this is done by adding `(content-type)` to the end of the `+ Response` line.

> *For Example: `+ Response 200 (application/json;charset=UTF-8)` means a Response Body with Content-Type `application/json;charset=UTF-8` that returns HTTP Status 200*
>
> ```
> + Response 200 (application/json;charset=UTF-8)
> ```

If the response body contains any data structures then you include these in the line underneath `+ Response`, for API blueprint to be rendered properly, the data/attributes should be indented by **four spaces** or **one TAB** character and is denoted using the `Attributes` field.

> ```
> + Response 200 (application/json;charset=UTF-8)
>
> 	+ Attributes (array[Furniture, Furniture])
> ```
>
> *Note: The response can be an Array of objects, a single object or no objects, be sure to choose the appropriate response type and http status code for your resource*

Sometimes, you want to include multiple responses for a single request, this is useful when the resource could return a custom error such as `ID Not Found`, to create multiple responses you simple create another `+ Response` section.

> ```
> + Response 201
>
> 	+ Attributes (Furniture)
>
> + Response 400
>
> 	+ Body
>
> 		{
> 		    "status": "ERROR",
> 		    "message": "Missing fields"
> 		}
> ```
>
> *Note: If your response **does not** use a data structure then you can omit the `Attributes` declaration and use the `Body` declaration instead to reference the JSON*

Additionally you are able to specify any additional headers to be sent on the response, to add custom headers in the Apiary document they should be indented by **twelve spaces** or **three TAB** characters from the `+ Response` heading and the headers are denoted using the `Headers` field.

```
+ Response 201

    + Headers

            example_header: 1234
```

Exercise 4.1: Modifying Request & Response Bodies

This exercise will introduce you to the Request and Response sections of the Apiary document.

> *Note: Refer back to the Specific Request Bodies and Specific Response Bodies sections for more information.*

Exercise 4.1 - Required Tools

- Internet Browser

Exercise 4.1 - Task

Currently the `Create` resource does not send or receive any fields as part of the request/response body, your task is to alter the request and response by adding the `Attributes` field so that the entire furniture object is passed in the request/response using the data structure you created in Exercise 3: Adding a Data Structure.

> *Tip: Remember the differences in syntax when using a `Data Structure` as opposed to a `JSON Object`*

Exercise 4.1 - Success Criteria

- ☐ Updated `Create` resource, `Request` section to include the entire Furniture object
- ☐ Updated `Create` resource, `Response` section to include the entire Furniture object
- ☐ Using the Console, ensure the request and response bodies contains the entire Furniture object
- ☐ Remove comments related to this exercise

Exercise 4.2: Adding Request and Response Bodies

This exercise introduces you to the concept of multi-request and multi-response resources.

Exercise 4.2 - Required Tools

- Internet Browser

Exercise 4.2 - Task

The `Create` resource currently showcases the "happy" path of an API, where the request is accepted and processed before returning with an object in the response. This exercise includes adding a un-happy path for a "bad" request that returns a particular error response.

In your service, it should return an error to the user if all of the fields of a Furniture object are not sent in the request.

Based on this requirement, your Request body should be missing the `id` and `name` fields, which should return a `Bad Request` error in the response with a suitable message/status combination.

Exercise 4.2 - Success Criteria

- [ ] New request body containing subset of the fields
- [ ] New response body containing message and status fields
- [ ] Bad Request response status code
- [ ] Using the Console, test out the new "un-happy" path for the `Create` resource
- [ ] Remove comments related to this exercise

Section 5 - Tying Everything Together

Exercise 5: Adding New Endpoints

In this final exercise, you will combine your learnings from the previous set of exercises in order to add a whole new endpoint to the Apiary document that includes request/response bodies with a new data structure.

Exercise 5 - Required Tools

- Internet Browser

Exercise 5 - Task

Based on your knowledge gained from earlier Exercises and the Introduction to Apiary section, your task is to create a new endpoint that handles the `Get By Id` resource.

This new endpoint should include a Path Parameter for the ID variable along with both a "happy" and "un-happy" path, an un-happy request can be classified if the ID of the furniture item is not found.

> *Tip: Don't forget to add the URI for the Get Furniture By Id resource*

Exercise 5 - Success Criteria

- [ ] New path parameter for ID
- [ ] "Happy" path Request and a suitable `OK` Response
- [ ] "Un-Happy" path Request and a suitable `Not Found` Response
- [ ] Understanding of how Apiary documents work
- [ ] Remove comments related to this exercise