Table of Contents

[Introduction 2](#_Toc506183941)

[Functional Requirement - FR 3](#_Toc506183942)

[Non Functional Requirement - NFR 4](#_Toc506183943)

[RAML specification:- 6](#_Toc506183944)

[Commentary on the interaction of use case 1 with the API 6](#_Toc506183945)

[Commentary on interaction of use case 2 with the API- 7](#_Toc506183946)

[Commentary on how the API could be extended for use case 3 8](#_Toc506183947)

[Commentary on any 'interesting' design decisions you made (and alternative options considered) 9](#_Toc506183948)

[Link to git repository where the code is stored along with the README.md 9](#_Toc506183949)

[API Service Call Details 10](#_Toc506183950)

[1. Customers - GET 11](#_Toc506183951)

[2. Customers – POST 12](#_Toc506183952)

[3. Customers - PUT 13](#_Toc506183953)

[4. Customers - DELETE 14](#_Toc506183954)

[5. Orders & Products Extensions– GET | POST | PUT | DELETE 15](#_Toc506183955)

[Design Principles and Non FR’s 16](#_Toc506183956)

[1. Versioning 16](#_Toc506183957)

[2. Paging 16](#_Toc506183958)

[3. Filtering 17](#_Toc506183959)

[4. Sorting 17](#_Toc506183960)

[5. Searching 17](#_Toc506183961)

[6. Partial answers (Field selection) 18](#_Toc506183962)

[7. Cross-domain (CORS) 18](#_Toc506183963)

[8. HATEOAS 18](#_Toc506183964)

[9. Non Resource scenarios 18](#_Toc506183965)

[10. Http Cache 19](#_Toc506183966)

[11. Compress Data 19](#_Toc506183967)

[Errors 20](#_Toc506183968)

[A. Structure 20](#_Toc506183969)

[B. Status Codes 20](#_Toc506183970)

[a. Success 20](#_Toc506183971)

[b. Client Error 21](#_Toc506183972)

[c. Server Error 22](#_Toc506183973)

[Security Overview 22](#_Toc506183974)

[Securing your API with OAuth2 22](#_Toc506183975)

[Solution Delivery 23](#_Toc506183976)

[API Summary 23](#_Toc506183977)

[Customer Service API 23](#_Toc506183978)

[Methods and sub-resources 24](#_Toc506183979)

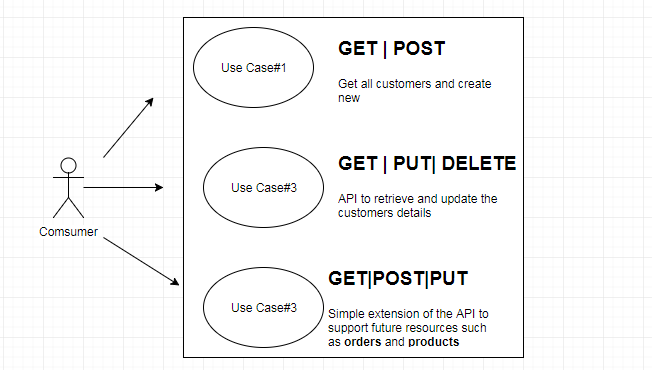
[Test Summary 24](#_Toc506183980)

# Introduction

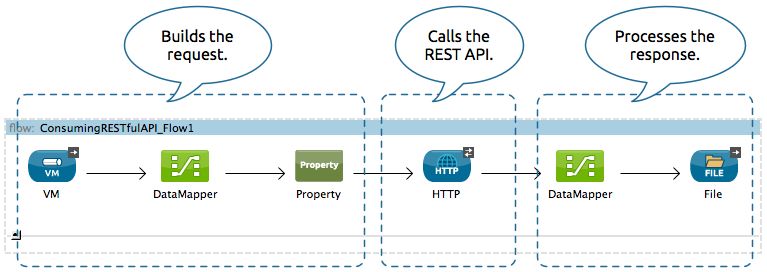
Design a RESTful API using RAML that contains a single resource, customers, and allows the following:

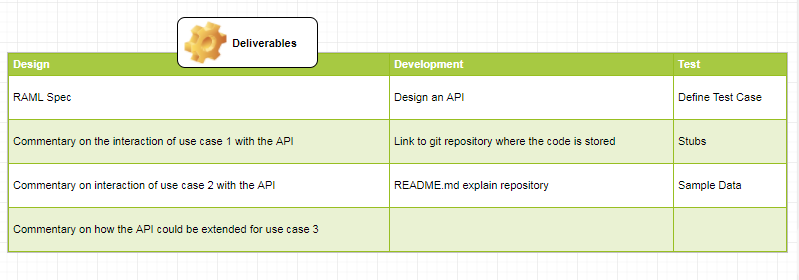
Service Action:-

1. List customers
2. Create a new customer
3. Update a customer
4. Deletes a customer



**Consuming REST API**





# Functional Requirement - FR

|  |
| --- |
| List customers – |
| Get All Customers |
| A consumer may periodically (every 5 minutes) consume the API to enable it (the consumer) to maintain a copy of the provider API's customers (the API represents the system of record)  FR\_API\_01 : List all customers |

|  |
| --- |
| List customer – |
| Get Customers |
| A mobile application used by customer service representatives that uses the API to retrieve and update the customer’s details.  FR\_API\_02 : Create a new customer  FR\_API\_03 : Update a customer  FR\_API\_04 : Deletes a customer |

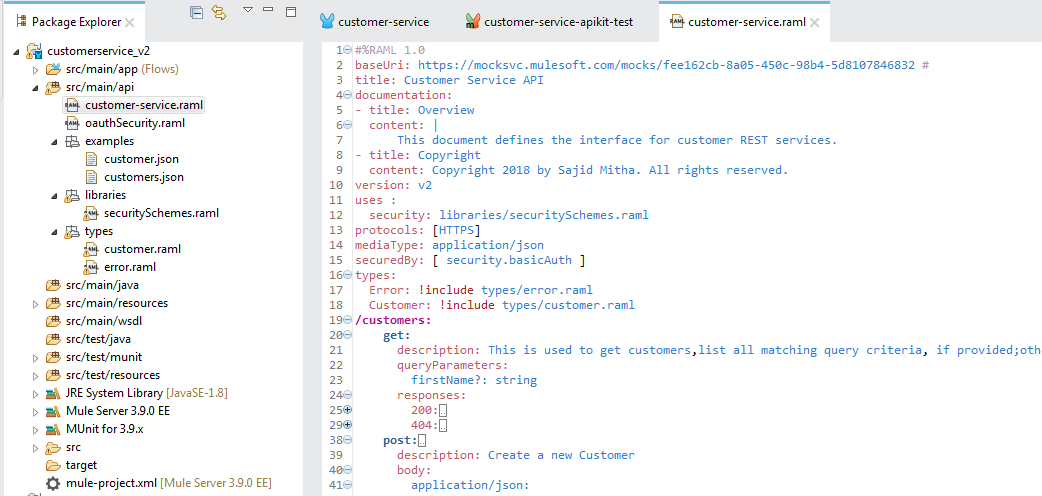
|  |
| --- |
| Extensions – |
| Orders & Products |
| Simple extension of Rest API to support future resources such as orders and products.  FR\_API\_05 : Extension to take care of future customers’ orders  FR\_API\_06 : Extension to take care of future customers’ products |

# Non Functional Requirement - NFR

|  |
| --- |
| List of requirements to support design– |
|  |
| Best practices for RESTful API design  NFR\_API\_01 :   1. API versioning, version api discussed in details in later section (/blog/api/v1) 2. The resource should always be plural in the API endpoint and if we want to access one instance of the resource, we can always pass the id in the URL. 3. Field name casing conventions, follow any casing convention, but make sure it is consistent across the application. If the request body or response type is JSON then please follow camelCase to maintain the consistency. 4. Use sub-resources for relations |
|  |
| Use of built in aspects of the HTTP protocol (status codes, headers etc.)  NFR\_API\_01 :   1. HTTP Status code to be used for all CRUD operations 2. Use HTTP headers for serialization formats  * Both, client and server, need to know which format is used for the communication. The format has to be specified in the HTTP-Header. * Content-Type defines the request format. * Accept defines a list of acceptable response formats.  1. The URL should only contain resources(nouns) not actions or verbs  * The API path/addCustomers contains the action addNew along with the resource name Customers * correct way, [HTTP methods](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods) (GET, POST, DELETE, PUT), also called as verbs, play the role..  1. GET method requests data from the resource 2. POST method requests the server to create a resource 3. PUT method requests the server to update resource 4. DELETE method requests that the resources 5. When the client raises a request to the server through an API, the client should know the feedback, whether it failed, passed or the request was wrong. HTTP status codes are bunch of standardized codes which has various explanations in various scenarios. The server should always return the right status code 6. Handle Errors with HTTP status codes  * It is hard to work with an API that ignores error handling. Pure returning of a HTTP 500 with a stack trace is not very helpful. * Use HTTP status codes, discuss in details in section:- Errors  1. Allow overriding HTTP method  * Some proxies support only POST and GET methods. To support a RESTful API with these limitations, the API needs a way to override the HTTP method. * Use the custom HTTP Header X-HTTP-Method-Override to override the POST Method. |
|  |
| Use of RAML features  NFR\_API\_01 :   1. The spec itself is written in YAML.YAML it is more concise and clutter-free 2. RAML ships with a bunch of features to keep our spec DRY (Don’t Repeat Yourself). 3. API specification with RAML helps defining some global characteristics like title, baseUri, supported protocols and the primary media type. 4. first line to tell the RAML parser the spec version 5. we define the endpoint by defining the /customers postfix which will be appended to the previously defined baseUri 6. The response body is specified as array of customer objects. Each user object contains the properties id, firstName,lastName and address with detailed description via primitives and enums. 7. Additionally, an example is provided which can be encoded as JSON which simplifies manual testing. 8. Usually, types are shared by different REST operations, endpoints or even APIs. RAML covers this requirement by extracting named type definitions. Types allow us to define a dedicated type for Customer, Error . Types allow to create associations, nesting, create hierarchies, introduce discriminators and many more. 9. Types also come with some syntactic sugar to keep the specs short and concise. Every named type is an object by default. Optional properties can be expressed by a trailing question mark and arrays have a shorthand syntax of [] 10. Traits enable to attach arbitrary behaviour and metadata similar to mixins. ResourceTypes are specialized traits to add behaviour with the ability to reference custom parameters and the resourcePath. We define the behaviour of the response 400 as BadRequest trait and the behaviour of the response 404 as NotFound resourceType. Traits are attached by using is while resourceTypes are attached using the type attribute. *We have not implemented this but its important to see if its applicable in future releases.* 11. Defined securitySchemes can be attached to endpoints and/or operations using securedBy. We set it just once to the root endpoint /customers to wrap everything underneath. 12. RAML provides different ways to modularize the spec by extracting particular parts to external files. In our example, we extract types and resourceTypes. 13. RAML 1.0 provides overlays and extensions are another great feature. Both allow us to refine our specification. This could be used to translate the API spec in different languages |
|  |
| Best Efficient use of the network (especially for use case 2)  NFR\_API\_01   1. Include paging to minimise data transfer. 2. Include partial answers to support minimise data transfer 3. Pointers to linked resource 4. Caching & Compression |
|  |
| Applicability of the API to the use cases (including edge and exceptional scenarios)  NFR\_API\_01 :   1. Take care of all available response type, leaving no assumptions for developers. |
|  |
| Breadth of considerations in the design  NFR\_API\_01 :   1. Authentication using Oauth2/Base 2. SSL for communication 3. Include optional features like Query parameters 4. Sorting, Search & Filtering 5. Support cross domain |
|  |
| Code quality (if optional task is attempted)  NFR\_API\_01 :   1. Git Hub link provided |
|  |

## RAML specification:-

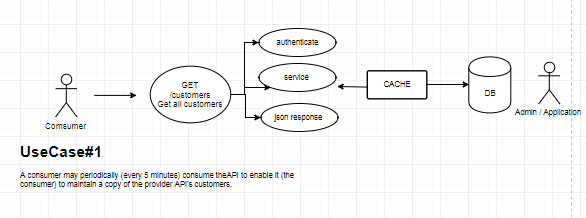
1. Attached customer-service.raml and related artefacts



## 

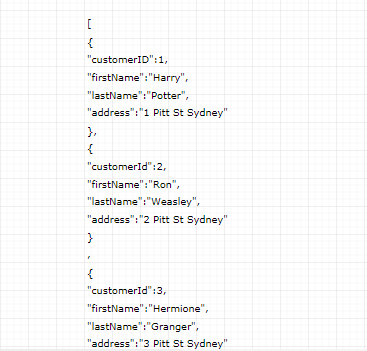
## Commentary on the interaction of use case 1 with the API

A consumer may periodically (every 5 minutes) consume the API to enable it (the consumer) to maintain a copy of the provider API's customers



Use case #1 is about a periodically scheduled API call to sync the client system with customer data that’s exposed via the API.

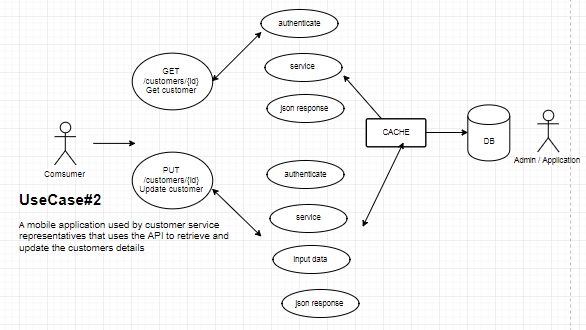
/customers - is the endpoint which this application/consumer would call on regular intervals which would provide it a json array of customers.



This request would be authenticated via Basic/Oauth2 security schema which is implemented by the API prior to execution of request.

We can have this request served via a Memcache as these calls are periodic and retrieves all data, and is IO intensive, thus to prevent continuous database hits we should cache it on server side.

## Commentary on interaction of use case 2 with the API-



Use case #2 is about an application used by customer service agents to retrieve customer details.

This is a stateless call to API to get a single customer data that’s exposed via the API using /customer/{id} endpoint.

/customers/{id} - is the endpoint which this application/consumer would call, which would provide it a json object of customer data.

This request would be authenticated via Basic/Oauth2 security schema which is implemented by the API prior to execution of request.

We can have this request served via a Memcache if these calls are frequent, thus prevent continuous database hits.

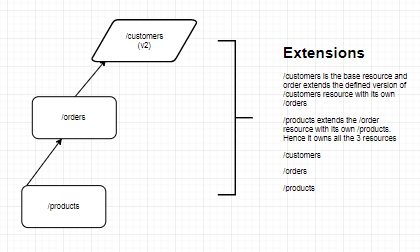
Since this service is used by mobile application its ideal to not show all available fields of customer object we should include partial answers to support and minimise data transfer, explained in detail later. For example if we can customise request by only for required fields.

GET /customers/007?fields=firstname,name

If possible we should use pointers to linked resource to give clarity and possible service calls.

Caching & Compression, if we can compress some data then it’s ideal to gzip while flowing on the network.

## Commentary on how the API could be extended for use case 3



*Overlays* and *extensions* are modules defined in external files that are used to extend an API. *An overlay* is used to extend non-behavioural aspects of an API, such as descriptions, usage directions, and user documentation items, whereas

An *extension* is used to extend or override behavioural aspects of the API.

In our case we have extended the feature list of the api by introducing 2 extensions.

/orders 🡪 extends (/customers)

Hence /orders endpoint share all what’s available in /customers + / orders

Similarly /products 🡪 extends (/orders)

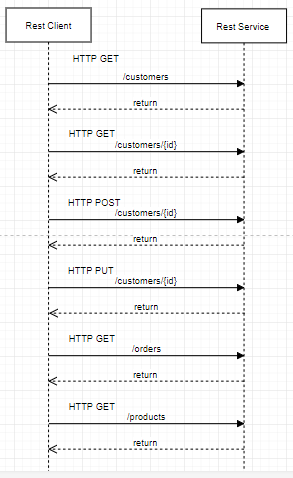
Assume /products is an extension of /orders hence in future if service needs to address /customers,/orders and /products this can be used.

## Commentary on any 'interesting' design decisions you made (and alternative options considered)

1. Have refactored API definition using include where necessary. The purpose of include is to modularize a complex property value in a RAML definition by placing the property value in an external file.
   1. data type for a Customer & Error object in a separate file
   2. example data for customer in a separate file
2. Have introduced BASIC & OAUTH authentication and datatype as a library. RAML libraries are used to modularize any number and combination of *data types, security schemes*, resource types, traits.

## Link to git repository where the code is stored along with the README.md

# API Service Call Details



Summary of API Calls:-

|  |  |  |  |
| --- | --- | --- | --- |
| HTTP | Action | Collection : /customers | Instance : /customers/{id} |
| GET | READ | Read a list of customers.  200 OK. | Read the detail of a single customer.  200 OK. |
| POST | CREATE | Create a new customer.  201 Created. | – |
| PUT | UPDATE | – | Full Update.  200 OK. Create a specific customer. 201 Created. |
| DELETE | DELETE | – | Delete order. 200 OK. |

## Customers - GET

|  |  |
| --- | --- |
| Title | *Get Customer* Example : Show All Customers or single customer  *Note: try to use verbs that match both request type (fetching vs modifying) and plurality (one vs multiple.). Add additional info here such as a description, if need be.* |
| URL | *The URL structure (path only, no root url)* **/customers** or  **/ customers /:id** or  **/ customers? Id=:id**  *For fixed urls: /customers*  *For urls with parameters in them: /customers/:id* |
| Method | *The request type* **GET** |
| URL Params | *If URL Params exist, specify them in accordance with name mentioned in URL section. Separate into optional and required.* Required:  None  Optional  firstName=[string]  example: firstName=Harry |
| Data Params | *None* |
| Success Response | *What should the status code be on success and is there any returned data? This is useful when people need to to know what their callbacks should expect!* Example:  Code: 200  Content: [{Customer Object}] |
| Error Response | *Most endpoints will have many ways they can fail. From unauthorized access, to wrongful parameters etc. All of those should be listed here. It might seem repetitive, but it helps prevent assumptions from being made where they shouldn’t be.* Example:  Code: 401 UNAUTHORIZED  Content: { error : "Log in" } OR Code: 404 Missing entry  Content: { code: “404” error : "Missing" } |
| Sample Call | *Just a sample call to your endpoint in a runnable format ($.ajax call or a curl request) – this makes life easier and more predictable.* $.ajax({  url: "/customers",  dataType: "json",  type : "GET",  success : function(r) {  console.log(r);  }  }); |
| Notes | *This is where all uncertainties, commentary, discussion etc. can go. I recommend timestamping and identifying oneself when leaving comments here.* |

## Customers – POST

|  |  |
| --- | --- |
| Title | *Get Customer* Example : Add A Customers  *Note: try to use verbs that match both request type (fetching vs modifying) and plurality (one vs multiple.). Add additional info here such as a description, if need be.* |
| URL | *The URL structure (path only, no root url)* **/ customers**  *For fixed urls: /customers* |
| Method | *The request type* **POST** |
| URL Params | *If URL Params exist, specify them in accordance with name mentioned in URL section. Separate into optional and required.* None |
| Data Params | *If making a post request, what should the body payload look like? Document your various data constraints too.* Example:  {  Customer : {  "id": "integer",  "firstName": "string",  "lastName": "string",  "address": "string"  }} |
| Success Response | *What should the status code be on success and is there any returned data? This is useful when people need to to know what their callbacks should expect!* Example:  Code: 201  Content: {  "message": "Customer created {but not really"  } |
| Error Response | *Most endpoints will have many ways they can fail. From unauthorized access, to wrongful parameters etc. All of those should be listed here. It might seem repetitive, but it helps prevent assumptions from being made where they shouldn’t be.* Example:  Code: 401 UNAUTHORIZED  Content: { error : "Log in" } OR Code: 404 Missing entry  Content: { code: “404” error : "Missing" } |
| Sample Call | *Just a sample call to your endpoint in a runnable format ($.ajax call or a curl request) – this makes life easier and more predictable.* $.ajax({  url: "/customers",  dataType: "json",  data : {  u: {  "id":5,  "firstName":"Severus",  "lastName":"Snape",  "address":"5 Pitt St Sydney"  }  },  type : "POST",  success : function(r) {  console.log(r);  }  }); |
| Notes | *This is where all uncertainties, commentary, discussion etc. can go. I recommend timestamping and identifying oneself when leaving comments here.* |

## Customers - PUT

|  |  |
| --- | --- |
| Title | *Update Customer* Example : update customer details  *Note: try to use verbs that match both request type (fetching vs modifying) and plurality (one vs multiple.). Add additional info here such as a description, if need be.* |
| URL | *The URL structure (path only, no root url)* **/ customers /:id** or  **/ customers? Id=:id**  *For fixed urls: /customers*  *For urls with parameters in them: /customers/:id* |
| Method | *The request type* **PUT** |
| URL Params | *If URL Params exist, specify them in accordance with name mentioned in URL section. Separate into optional and required.* Required:  id=[integer]  example: id=1 |
| Data Params | *If making a post request, what should the body payload look like? Document your various data constraints too.* Example:  {  Customer : {  "id": "integer",  "firstName": "string",  "lastName": "string",  "address": "string"  }} |
| Success Response | *What should the status code be on success and is there any returned data? This is useful when people need to to know what their callbacks should expect!* Example:  Code: 201  Content: {  "message": "Customer updated {but not really"  } |
| Error Response | *Most endpoints will have many ways they can fail. From unauthorized access, to wrongful parameters etc. All of those should be listed here. It might seem repetitive, but it helps prevent assumptions from being made where they shouldn’t be.* Example:  Code: 401 UNAUTHORIZED  Content: { error : "Log in" } OR Code: 404 Missing entry  Content: { code: “404” error : "Missing" } |
| Sample Call | *Just a sample call to your endpoint in a runnable format ($.ajax call or a curl request) – this makes life easier and more predictable.* $.ajax({  url: "/customers",  dataType: "json",  data : {  u: {  "id":5,  "firstName":"Severus",  "lastName":"Snape",  "address":"5 Pitt St Sydney"  },  type : "PUT",  success : function(r) {  console.log(r);  }  }); |
| Notes | *This is where all uncertainties, commentary, discussion etc. can go. I recommend timestamping and identifying oneself when leaving comments here.* |

## Customers - DELETE

|  |  |
| --- | --- |
| Title | *Delete Customer* Example : Delete a customer  *Note: try to use verbs that match both request type (fetching vs modifying) and plurality (one vs multiple.). Add additional info here such as a description, if need be.* |
| URL | *The URL structure (path only, no root url)* **/ customers /:id** or  **/ customers? Id=:id**  *For fixed urls: /customers*  *For urls with parameters in them: /customers/:id* |
| Method | *The request type* **GET** | **POST** | **DELETE** | **PUT** |
| URL Params | *If URL Params exist, specify them in accordance with name mentioned in URL section. Separate into optional and required.* Required:  id=[integer]  example: id=1 |
| Data Params | *If making a post request, what should the body payload look like? Document your various data constraints too.* Example: None |
| Success Response | *What should the status code be on success and is there any returned data? This is useful when people need to to know what their callbacks should expect!* Example:  Code: 201 Content: {  "message": "Customer deleted {but not really"  } |
| Error Response | *Most endpoints will have many ways they can fail. From unauthorized access, to wrongful parameters etc. All of those should be listed here. It might seem repetitive, but it helps prevent assumptions from being made where they shouldn’t be.* Example:  Code: 401 UNAUTHORIZED  Content: { error : "Log in" } OR Code: 404 Missing entry  Content: { code: “404” error : "Missing" } |
| Sample Call | *Just a sample call to your endpoint in a runnable format ($.ajax call or a curl request) – this makes life easier and more predictable.* $.ajax({  url: "/customers",  dataType: "json",  type : "DELETE",  success : function(r) {  console.log(r);  }  }); |
| Notes | *This is where all uncertainties, commentary, discussion etc. can go. I recommend timestamping and identifying oneself when leaving comments here.* |

## Orders & Products Extensions– GET | POST | PUT | DELETE

|  |  |
| --- | --- |
| Title | *Get Orders or Get Products* Example : Show All Orders or Products  *Note:* These are extensions to the customer api and made ready for future versions*.* |
| URL | *The URL structure (path only, no root url)* **/order or**  **/ order /:id or**  **/products**  **/ products/id**  *For fixed urls: /orders*  *For urls with parameters in them: /orders/:id* |
| Method | *The request type* **GET** | **POST** | **DELETE** | **PUT** |
| URL Params | *If URL Params exist, specify them in accordance with name mentioned in URL section. Separate into optional and required.* Required:  id=[integer]  example: id=12  Optional  size=[numeric] /\* to show pagination \*/  example: size=20 |
| Data Params | *If making a post request, what should the body payload look like? Document your various data constraints too.* Example: |
| Success Response | *What should the status code be on success and is there any returned data? This is useful when people need to to know what their callbacks should expect!* Example:  GET Code: 200  Content: { id : 1 }  POST/PUT/DELETE  Code: 201  Content: { id : 1 } |
| Error Response | *Most endpoints will have many ways they can fail. From unauthorized access, to wrongful parameters etc. All of those should be listed here. It might seem repetitive, but it helps prevent assumptions from being made where they shouldn’t be.* Example:  Code: 401 UNAUTHORIZED  Content: { error : "Log in" } OR Code: 404 Missing entry  Content: { code: “404” error : "Missing" } |
| Sample Call | *Just a sample call to your endpoint in a runnable format ($.ajax call or a curl request) – this makes life easier and more predictable.* $.ajax({  url: "/orders",  dataType: "json"  type : "GET",  success : function(r) {  console.log(r);  }  }); |
| Notes | *This is where all uncertainties, commentary, discussion etc. can go. I recommend timestamping and identifying oneself when leaving comments here.* |

# Design Principles and Non FR’s

## Versioning

API will have to evolve over time. There are several ways of versioning an API:

Suggested approach:-

* In the*path*, at the beginning or at the end of the *URI (Preferred)*
  + [http://customerservicev2.cloudhub.io/api/v1/customers](https://localhost/customerapi/v1/customers)
  + [http://customerservicev2.cloudhub.io/api/v2/customers/007](https://localhost/customerapi/v2/customers/007)

In a *HTTP Header*

E.g.:- Login Header

|  |  |  |
| --- | --- | --- |
| Type | Params | Values |
| HEAD  HEAD | authkey  version | string  number |

## Paging

Paging - default values when they are not provided by the calling client, for example with a range of values [0-25]. Pagination is often managed in the URL through the *query-string*. HTTP headers also provide this mechanism. Propose that we use a range of values through your collection’s resources index. As an example, resources from index 10 to 25 included is equivalent to */customers?range=10-25*

Note:

If 25 customers are requested of the 48 available, we receive a 206 Partial Content return code.

## Filtering

Filtering consists in restricting the number of queried resources by specifying some attributes and their expected values. It is possible to filter a collection on several attributes at the same time, and to allow several values for one filtered attribute.

We propose to use directly the attribute’s name with an equal sign and the expected values, each of them separated by a comma.

Example:

Retrieve customers with first name as John

[http://customerservicev2.cloudhub.io/api/v1/customers?fname= John](%20http://customerservicev2.cloudhub.io/api/v1/customers?fname=%20John%20)

Retrieve customers with first name as John & Mark

http://customerservicev2.cloudhub.io/api/v1/customers?fname= John,Mark&lname=Tate

## Sorting

Sorting the result of a query on a collection of resources requires two main parameters:

**sort**: Contains the names of the attributes on which the sorting is performed, separated by a comma.

**desc**: By default, the sorting is done in ascending order. If one wishes to sort in descending order, they need to add this parameter (without any value). In some specific cases, one may want to specify which attributes should be used as ascending sort keys and which as descending sort keys. Then, the desc parameter should contain the attributes that will be descending sort keys, the others will be ascending sort keys.

Example: retrieving the list of restaurants sorted by name

## Searching

If filtering does not fit our needs (to make partial or approximate matches, for instance), we need the ability to search the available resources.

A search is a sub-resource of our collection. As such, its results will have a different format than the resources and the collection itself. This allows us to add suggestions, corrections and information related to the search.

Example

https:// customerservicev2.cloudhub.io/api/customers/search?name=la\*

Global search should have the same behavior as resource-specific search, except that it is located at the root of the API and therefore must be pointed out in the documentation.

https:// customerservicev2.cloudhub.io/api/search?name=John

## Partial answers (Field selection)

Partial answers allow clients to retrieve only the information they need. This feature is vital in mobile contexts (UMTS-) where bandwidth usage must be optimized.

Recommend at least being able to select the attributes to be retrieved, over 1 level of resource, through the Google notation fields=attribute1, attributeN.

Example:-

GET /customers/007?fields=firstName,lastName

200 OK { "id":"007", "firstName":"James", "lastName":"Bond" }

GET /clients/007?fields= firstName, lastName,address

200 OK

{

"id":"007",

"firstname":"James",

"name":"Bond",

"address":{"street":"Horsen Ferry Road"}

}

## Cross-domain (CORS)

When the application (JavaScript SPA) and the API is hosted on different domains, for example:

http://customerservicev2.cloudhub.io

http://api.customerservicev2.cloudhub.io

A good practice consists in using the [CORS](http://en.wikipedia.org/wiki/Cross-origin_resource_sharing) protocol which is the HTTP standard.

On the client side, implementation is imperceptible: the browser will send a HTTP request with the OPTIONS verb before any GET/POST/PUT/PATCH/DELETE request.

## HATEOAS

A call to /customers/007 would then return the details of the customer, along with pointers towards linked resources:

GET /customers/007 <

200 Ok

{ "id":"007", "firstname":"James",”Bond”,”MI6 London”, "links": [{"rel":"products","href":"https:// localhost /v1/ products /42", "method”: “GET"}, {"rel":"orders", "href":"https://localhost/v1/orders/1234", "method”: “GET"}, ] }

## Non Resource scenarios

In theory, any request must be seen and manipulated as a resource. In real life, it’s not always possible

To design properly this exception in your API, the simplest solution is to consider that any POST request is an action with an implicit or explicit verb.

For a collection of entity resources for instance, the default action is a creation:

POST /customers/12

< 201 OK

< { "id": 12 }

## Http Cache

Note: It’s only for HTTP GET requests!

So let’s say that the client sends a request for some metadata, and we want the client to cache it for 1 hour. This might reduce network traffic but it needs a valid use case.

GET /customers/metadata HTTP/1.1

Host: customerservicev2.cloudhub.io

Accept: application/json

Accept-Language: en

To do this, we just add the **Cache-Control** header to our response:

HTTP/1.1 200 OK

Content-Type: application/json

**Cache-Control: max-age=3600**

Content-Length: 88

Etag: "6d82cbb050ddc7fa9cbb659014546e59"

{

"languageCodes": [

{"da":"Danish"},

{"no":"Norwegian"},

{"en":"English"}

]

}

## Compress Data

In addition to reducing payload size by using [JSON](http://www.json.org/), optimize by compressing data. We recommend using [gZIP](https://www.gnu.org/software/gzip/) for file compression and decompression of large messages.

The only reason when you wouldn’t want to use gZIP is when you have many small endpoints that provide very small amounts of data

# Errors

## Structure

Implement following JSON structure:- You can extend with more details, but this is minimum required.

{

"message" : "short\_description",

"code" : “error code”

}

## Status Codes

We are using HTTP return codes, as a code exists for every common case, which everybody understands..

### Success

200 OK is the usual success code for most cases. It is especially used when the first GET request on a resource is successful.

### Client Error

|  |  |
| --- | --- |
| TTP Status | Description |
| 400 Bad Request | Commonly used for calling errors if no other status matches. We can distinguish between two error types: Request behaviour error  GET /customers?caller=1  < 400 Bad Request  < {"error": "invalid\_request", "error\_description": "There is no ‘caller property on customers."}  Application condition error  POST /customers  {"firstName":"John "}  < 400 Bad Request  < {"error": "invalid\_user", "error\_description": "A user must have an id"} |
| 401 Unauthorized | I do not know your id. Tell me who you are and I will check your authorizations.   |  |  | | --- | --- | | 1 2 3 | GET /customers/4  < 401 Unauthorized < {“error”: “no\_credentials”, “error\_description”: “This resource requires authorization, you must be authenticated and have the correct rights to access it” } | |
| 403 Forbidden | You are identified, but you do not have the necessary authorizations.  GET /customers/4  < 403 Forbidden  < {"error": "not\_allowed", "error\_description": "You're not allowed to perform this request"} |
| 404 Not Found | The resource you asked for does not exist.  GET /customers/999999/  < 400 Not Found  < {"error": "not\_found", "error\_description": "The customer with the id ‘999999' doesn't exist" } |
| 405 Method not allowed | Either calling a method on this resource has no meaning, or the user is not authorized to make this call.  POST /customers/8000  < 405 Method Not Allowed  < {"error":"method\_does\_not\_make\_sense", "error\_description":"How would you even post a person?"} |

### Server Error

HTTP Status description

|  |  |
| --- | --- |
| 500 Server error | This request is correct, but an execution problem has been encountered. The client cannot really do much about this. We recommend to systematically return a Status 500.  GET /customers  < 500 Internal server error  < Content-Type: application/json  < {"error":”server\_error", "error\_description":"Oops! Something went wrong..."} |

# Security Overview

## Securing your API with OAuth2

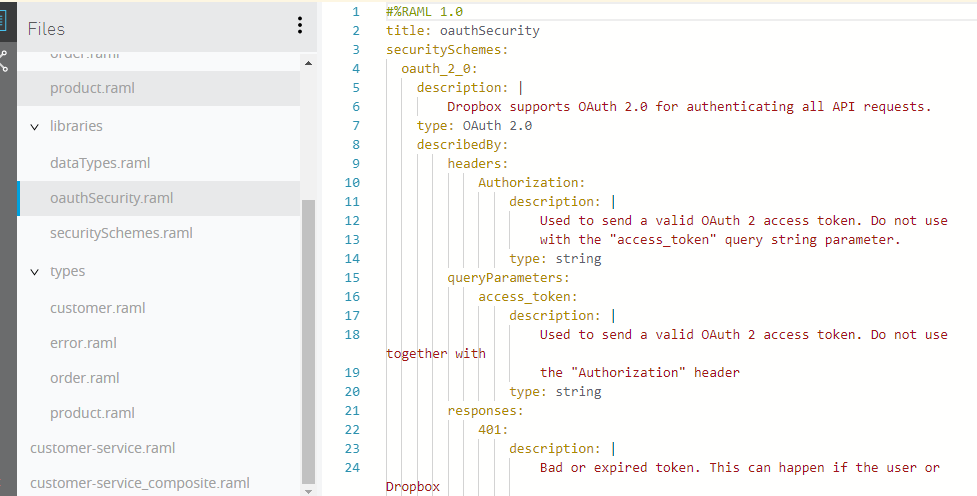
Always use HTTPS when communicating with:

1. OAuth2 providers
2. API providers

* Have done the prototype with BASIC authentication, but have attached both security schemes in the library.
* Select OAuth2 for authentication as it allows us to manage authentication and resource authorization for any type of application (native mobile app, native tablet app, JavaScript app, server side web app, batch processing…) with or without the resource owner’s consent.
* OAuth2 is also the de facto standard for securing APIs.

 Validate OAuth2 implementation; you might want to try the following test:

* Develop a client consuming your OAuth2 implementation and make a call to your API
* Then, replace the domain names of your API with Google’s API domain names.



# Solution Delivery

## API Summary

|  |  |
| --- | --- |
| API summary | Customer Service API Version:v2  Supported media type:application/json  Supported protocols:HTTPS API base URI https://mocksvc.mulesoft.com/mocks/12bb4c53-de2f-42a4-a16f-dad240c9e511 API resources [/customers](https://anypoint.mulesoft.com/designcenter/designer/) |
| **Types** |  |
| Error | Type Error ***Examples***  {  "code": "integer",  "message": "string"  } |
| Customer | Type Customer ***Examples***  {  "id": "integer",  "firstName": "string",  "lastName": "string",  "address": "string"  } |
| **Resources** |  |
| /customers | Methods and sub-resources Methods  [get](https://anypoint.mulesoft.com/designcenter/designer/)  This is used to get customers, list all matching query criteria, if provided; otherwise list all Customers  [post](https://anypoint.mulesoft.com/designcenter/designer/)  Create a new Customer  Sub-resources  [/{id}](https://anypoint.mulesoft.com/designcenter/designer/)  [/search](https://anypoint.mulesoft.com/designcenter/designer/) |
| GET |
| POST |
| /{id} |
| GETPUTDELETE |
| /orders | Future Resources or Extension |
| GET |  |
| /products | Future Resources or Extension |
| GET |  |

## Test Summary

Build Munit Test cases to test all positive and error scenarios, build cases for all possible scenarios.

**Sample:-**

