Article
Istvan Hahn · Oct 12, 2016 12m read

RESTful API

Beginner ' s guide to RESTful Application Program Interface (API) design and documentation. Through the example you will learn some common pattern for RESTful API.

Before you read

You need to know

- How to create RESTful web service in Ensemble
- How to consume RESTful web service in Ensemble
- How to pass service parameters
- How to return service result

What is a Service API?

What is an Application Programming Interface? Is it something materialized? Is it a single programming unit? What is an API for? From my point of view an API is something which is determined by the program code in an indirect way. But the fully defined API is provided by a container (controlled by deployment settings) running the executable program. So I rather define the API as the public description of a service. The description could be either human readable or for robots only. Or both. An API is for sharing essential information about a service with those who are about to consume it. An API explains what the service for, the context in which it can be used, what the functions are, what data structures are managed etc.

In the good old days the " program documentation " was more or less " a necessary evil " . The modern programming languages were forcing kind of a documentation by introducing declarations in the program source. Although declarations were the " robot " readable documentation, by using tools (runoff, Java doc…) you could extract information and format it for humans. Even if no single line of true documentation was add to the source, these tools were still able to produce some minimal text.

Is it somewhat different these days? Not really. Service API remained an abstraction what people mean by collection of information required to properly use a functional piece of computer software. There are languages to formalize the API definition like Web Services Description Language (WSDL). Unfortunately those have limited use. Not because for example WSDL is not capable enough to express a RESTful API, but because of non-technical mismatch. (How does it look like expressing a JSON structure in XML?) At the end there is no such de-facto standard language for REST as WSDL for SOAP Web Services. Pity. Is not it?

Never mind. Anyhow, first we need to understand what an API is documenting.

What an API is made of?

What are the core attributes of a service.

- Service location. The URL root path of the service. Like http://localhost:57774/csp/msa/person.
- Service methods. Those are the functions of a service. A method is defined by combination of the verb from

the HTTP header (GET, POST, PUT…) and the additional path type parameters.
- Accepted method parameters. A list of parameter with its type. The type can be **path** for parameters included in the URL path; **query** for URL query encoded; **form** for form data; **content** for HTTP message body.
- Return status. The status field from the HTTP response header. There might be several return status codes per service method. The number is depending on the service method and the granularity of exception handling.
- Response contents. The expected contents per status code. The format can be different by status code. For example on successful completion of a request a JSON serialized object is expected. In case of server error (500) a plain text explanation is sent.

Example API

First let us try do describe what are we trying to achieve. We are going to build a very simple service. A registry service. It is going to manage resources of the same type. For example persons.

The structure is very simple: a name, date and place of birth, mother ' s maiden name and a generated internal unique registry ID. Place has a structure like: country, city.

We would like to insert a new record into our registry (update full registry entry), update individual attributes of an entry (update attribute), delete an entry, get a single entry by registry ID, query a list of registry IDs based on attribute match.

We also want some service function: initialize a registry, populate some records for testing.

From the external world we would like http://localhost:57774/csp/msa/person as service location.

Adding a new entry via **PUT** on the service location. Sending the registry record as a content. On return the complete entry with registry ID is expected.

Update via **POST**. The URL is complete with the registry ID of the entry to be updated. The attributes to be updated are sent as form data.

GET is used to retrieve data. If the URL path ends with an ID, than the registry entry identified by the ID is returned. If no ID found but there is a query in the URL, than a list of IDs is returned. For example http://localhost:57774/csp/msa/person/12A33 returns the entry 12A33. The query key value pairs are the attribute matching clauses used internally for selecting the entries. For example http://localhost:57774/csp/msa/person?name=Hahn%20Istvan&dob=1961 returns a list of persons who was born in 1961 with the name Hahn Istvan.

DELETE does delete.

POST to http://localhost:57774/csp/msa/person/init will initialize the registry.

POST to http://localhost:57774/csp/msa/person/populate/100 loads 100 of test entries.

API Documentation

The following section gives an example how the service API could be documented. Please remember, that neither the structure nor the contents are standardized. It is just an example.

I tried to make the documentation " tool agnostic " . There are documentation tools on the market. Some of them do their job almost as good as it ought to be. The intent of this section is to give you a feeling what the complexity of an API documentation is if you do it with a text editor.

Resource: person

A generic service to manage person type resources. The person has a minimum set of attributes. Basically demographics and a registry ID.

Location: http://localhost:57774/csp/msa/person

Method:

Get a single resource based on the unique ID of the resource.

Verb: GET

Parameters:

Name	Type	Data type	Comment
1	Path	Resource ID	Unique ID of the resource to be retrieved.

Response:

Status	Return type	Comment
200	Person	Record found.
204	None	No record with Resource ID exists.
401	None	Unauthorized access. The resource needs user credential in the header.
403	None	Forbidden. User is not authorized to access the resource.
500	Error	Internal server error.
501	Error	Requested method is not implemented.
503	Error	Service is temporarily unavailable.

Method:

Get a list of matching resource IDs based on non-unique query. The method uses the query part to build the query string. The query key/ value pairs are translated to column name/ value pairs.

Verb: GET

Parameters:

Name	Type	Data type	Comment
name	query	string	Search criteria.
motherMaidenName	Query	String	
dob	Query	Date	
birthPlaceCounty	Query	String	
birthPlaceCity	Query	string	

Response:

Status	Return type	Comment
200	Person	Record found.
204	None	No matching record.
401	None	Unauthorized access. The resource needs user credential in the header.
403	None	Forbidden. User is not authorized to access the resource.
500	Error	Internal server error.
501	Error	Requested method is not implemented.
503	Error	Service is temporarily unavailable.

Method:

Delete an entry from the registry.

Verb: DELETE

Parameters:

Name	Type	Data type	Comment
1	path	string	Unique registry ID.

Response:

Status	Return type	Comment
200	Person	Record deleted.
401	None	Unauthorized access. The resource needs user credential in the header.
403	None	Forbidden. User is not authorized to access the resource.
500	Error	Internal server error.
501	Error	Requested method is not implemented.
503	Error	Service is temporarily unavailable.

Method:

Add or update an entry to the registry.

Verb: PUT

Parameters:

Name	Type	Data type	Comment
None	content	JSON	An object serialized to JSON format.

Response:

Status	Return type	Comment
200	Person	The entry with the generated resource ID either newly add or updated by the registry service.
401	None	Unauthorized access. The resource needs user credential in the header.
403	None	Forbidden. User is not authorized to access the resource.
500	Error	Internal server error.
501	Error	Requested method is not implemented.
503	Error	Service is temporarily unavailable.

Method:

Update individual attributes of a registry entry.

Verb: POST

Parameters:

Name	Type	Data type	Comment	
1	Path	String	The resource ID of the entry to be updated.	
name	Form	string	New value of the attribute	
motherMaidenName	Form	String		
dob	Form	Date		
birthPlaceCounty	Form	String		
birthPlaceCity	Form	string		

Response:

Status	Return type	Comment
200	Person	Record updated.
204	None	No record with Resource ID exists.

401	None	Unauthorized access. The resource needs user credential in the header.
403	None	Forbidden. User is not authorized to access the resource.
500	Error	Internal server error.
501	Error	Requested method is not implemented.
503	Error	Service is temporarily unavailable.

Method:

Initialize the registry.

Verb: POST

Parameters:

Name	Type	Data type	Comment
init	Path		

Response:

Status	Return type	Comment
200	None	Initialized.
401	None	Unauthorized access. The resource needs user credential in the header.
403	None	Forbidden. User is not authorized to access the resource.
500	Error	Internal server error.
501	Error	Requested method is not implemented.
503	Error	Service is temporarily unavailable.

Method:

Populate test data.

Verb: POST

Parameters:

Name	Type	Data type	Comment
populate	Path		
2	Path	Numeric	Number of entries to be populated.

Response:

Status	Return type	Comment
200	None	Initialized.
401	None	Unauthorized access. The resource needs user credential in the header.
403	None	Forbidden. User is not authorized to access the resource.
500	Error	Internal server error.
501	Error	Requested method is not implemented.
503	Error	Service is temporarily unavailable.

Data structures:

Person

Name	Type	Flag	Comment
ID	RegistryID	R	Generated registry
Name	String	R	Name of the perso native language fo
DOB	Date	R	Date of birth.
BirthPlace	BirthPlace	O	Birth place.
MotherMaidenName	String	O	Mother ’ s maiden

BirthPlace

Name	Type	Flag	Comment
Country	String	O	Country code.
City	String	R	City name

Error

Name	Type	Flag	Comment
Code	String	R	Error code
Text	String	O	Error text

Name	Type	Flag	Comment
InnerError	Error	O	An inner error repo subcomponent of t component.

Implementation

The following section gives an example for the resource registry we discussed earlier. This is (again) just an example.

To make you understand easier, I group the source in an artificial way.

n Everything belonging to the API is squeezed into the resource map class.

n The complete UrlMap XData block is sliced into single **Route** entries.

n Each entry is glued to the static method actually implementing the functionality.

So to recover the true class needs some (re-)engineering. Please, happy (re-)engineering!

The first service method is the query…

```
<!-- Query the registry. The URL query part holds select criteria. -->

<Route Url="/:service" Method="GET" Call="QueryRegistry"/>

classmethod QueryRegistry(service) as %Status {

        try {

                set serviceInstance = ..getServiceInstance(service)

                do
 ..dumpResponse(serviceInstance.runQuery(..getQueryParameters($listbuild("name","dob"
,"motherMaidenName","birthPlaceCountry","birthPlaceCity"))))

        }

        catch ex {

                do ..ReportHttpStatusCode(..getHTTPStatusCode(ex),ex.AsStatus())

        }

        quit $$$OK

}
```

```
<!-- Get a single entry -->
<Route Url="/:service/:registryID" Method="GET" Call="GetEntry"/>
classmethod GetEntry(service,registryID) as %Status {
        try {
                set serviceInstance = ..getServiceInstance(service)
                do ..dumpResponse(serviceInstance.get(registryID))
        }
        catch ex {
                do ..ReportHttpStatusCode(..getHTTPStatusCode(ex),ex.AsStatus())
        }
        quit $$$OK
}
```

```
<!-- Delete a single entry -->
<Route Url="/:service/:registryID" Method="DELETE" Call="DeleteEntry"/>
classmethod DeleteEntry(service,registryID) as %Status {
        try {
                set serviceInstance = ..getServiceInstance(service)
                do ..dumpResponse(serviceInstance.delete(registryID))
        }
        catch ex {
                do ..ReportHttpStatusCode(..getHTTPStatusCode(ex),ex.AsStatus())
        }
        quit $$$OK
}
```

```
<!-- Utility method to initialize the registry. -->
<Route Url="/:service/_init" Method="POST" Call="InitializeRegistry"/>
classmethod InitializeRegistry(service) as %Status {
        try {
                set serviceInstance = ..getServiceInstance(service)
                do ..dumpResponse(serviceInstance.init())
        }
        catch ex {
                do ..ReportHttpStatusCode(..getHTTPStatusCode(ex),ex.AsStatus())
        }
        quit $$$OK
}

<!-- Utility method to populate test data. -->
<Route Url="/:service/_populate/:numberOfRecords" Method="POST" Call="Populate"/>
classmethod Populate(service,numberOfRecords) as %Status {
        try {
                set serviceInstance = ..getServiceInstance(service)
                do ..dumpResponse(serviceInstance.populate(numberOfRecords))
        }
        catch ex {
                do ..ReportHttpStatusCode(..getHTTPStatusCode(ex),ex.AsStatus())
        }
        quit $$$OK
}
```

```
<!-- Update individual attributes of a registry entry. -->

<Route Url="/:service/:registryID" Method="POST" Call="UpdateAttribute"/>

classmethod UpdateAttribute(service,registryID) as %Status {

        try {

                set serviceInstance = ..getServiceInstance(service)

                do
 ..dumpResponse(serviceInstance.updateAttribute(registryID, ..getFormParameters($list
build("name","dob","motherMaidenName","birthPlaceCountry","birthPlaceCity"))))

        }

        catch ex {

                do ..ReportHttpStatusCode(..getHTTPStatusCode(ex),ex.AsStatus())

        }

        quit $$$OK

}




<!-- Add a new or update an existing registry entry. -->

<Route Url="/:service" Method="PUT" Call="AddOrUpdate"/>

classmethod AddOrUpdate(service) as %Status {

        try {

                set serviceInstance = ..getServiceInstance(service)

                do
 ..dumpResponse(serviceInstance.addOrUpdate(..getContentParameter()))

        }

        catch ex {

                do ..ReportHttpStatusCode(..getHTTPStatusCode(ex),ex.AsStatus())

        }
```

```
        quit $$$OK

}
```

Now this is time to share you the utility methods.

```
classmethod getServiceInstance(serviceName) as Ens.BusinessService {

        set
 status = ##class(Ens.Director).CreateBusinessService(serviceName, .instance)

        throw:$$$ISERR(status) ##class(NoProduction).%New(status)

        quit instance

}



classmethod getHTTPStatusCode(ex) {

        quit $case(ex.%ClassName(1),

                              ##class(NoProduction).%ClassName(1)                  :5
03,

                              ##class(NotImplemented).%ClassName(1)        :501,


                             :500)

}



classmethod dumpResponse(responseObject) {

        if $isObject(responseObject) {

                if
 responseObject.%Ex
tends(##class(%DynamicObject).%ClassName(1)) { write responseObject.%ToJSON() }

                elseif
 responseObject.%Extends(##class(%ZEN.proxyObject).%ClassName(1)) {

                        do
 ##class(%ZEN.Auxiliary.jsonProvider).%ObjectToJSON(responseObject)

                }

                elseif responseObject.%Extends(##class(%XML.Adaptor).%ClassName(1)) {
```

```
                        do responseObject.XMLExportToString(.ret)
                        write ret
                }
                else { throw ##class(Serialization).%New() }
        }
        else {
                write responseObject
        }
}


classmethod getQueryParameters(parameterList) as %DynamicObject {
        set parameterObject = {}
        for i=1:1:$listlength(parameterList) {
                set parameterName=$listget(parameterList,i)
                set
 $property(parameterObject, parameterName) = %request.Get(parameterName)
        }
        quit parameterObject
}


classmethod getFormParameters(parameterList,queryObject) as %DynamicObject {
        if $data(queryObject) { set parameterObject = queryObject }
        else { set parameterObject = {} }
        for i=1:1:$listlength(parameterList) {
                set parameterName=$listget(parameterList,i)
                set
 $property(parameterObject, parameterName) = %request.Get(parameterName)
        }
        quit parameterObject
```

```
}
```

```
classmethod getContentParameter() as %DynamicObject {

        quit {}.%FromJSON(%request.Content)

}
```

And this is the end. We finished designing(?), implementing and documenting a RESTful Web Service API.

Stay tuned, I ' ll be back soon with further reading on Ensemble RESTful web services. The next is " Creating an Ensemble MicroService using RESTful Web Services " .

#Beginner #REST API #Ensemble

Source URL:https://community.intersystems.com/post/restful-api