Article
Sean Connelly · Sep 10, 2019 18m read

Open Exchange

Creating a OData API Adapter for InterSystems IRIS

In this article, we will explore the development of an IRIS client for consuming RESTful API services that have been developed to the OData API standard.

We will be exploring a number of built-in IRIS libraries for making HTTP requests, reading and writing to JSON payloads, and seeing how we can use them in combination to build a generic client adaptor for OData. We will also explore the new JSON adapter for deserializing JSON into persistent objects.

Working with RESTful APIs

REST is a set of engineering principles that were forged from the work on standardizing the world wide web. These principles can be applied to any client-server communication and are often used to describe an HTTP API as being RESTful.

REST covers a number of broad principles that include stateless requests, caching, and uniform API design. It does not cover implementation details and there are no general API specifications to fill in these gaps.

The side effect of this ambiguity is that RESTful APIs can lack some of the understanding, tools, and libraries that often build up around stricter ecosystems. In particular, developers must construct their own solutions for the discovery and documentation of RESTful APIs.

OData

OData is an OASIS specification for building consistent RESTful API's. The OASIS community is formed from a range of well-known software companies that include Microsoft, Citrix, IBM, Red Hat, and SAP. OData 1.0 was first introduced back in 2007, and the most recent version 4.1 was released this year.

The OData specification covers things like metadata, consistent implementations of operations, queries, and exception handling. It also includes additional features such as actions and functions.

Exploring the TripPinWS OData API

For this article we ' ll be using the TripPinWS API, which is provided as an example by Odata.org.

As with any RESTful API, we would typically expect a base URL for the service. Visiting this base URL in OData will also return a list of API entities.

https://services.odata.org:443/V4/(S(jndgbgy2tbu1vjtzyoei2w3e))/TripPinServiceRW

We can see that the API includes entities for Photos, People, Airlines, Airports, Me, and a function called GetNearestAirport.

The response also includes a link to the TripPinWS metadata document.

https://services.odata.org/V4/(S(djd3m5kuh00oyluof2chahw0))/TripPinServiceRW/$metadata

The metadata is implemented as an XML document and includes its own XSD document. This opens up the possibility of consuming metadata documents using code generated from the IRIS XML schema wizard.

The metadata document might look fairly involved at first glance, but it's just describing the properties of types that are used to construct entity schema definitions.

We can get back a list of People from the API by using the following URL.

https://services.odata.org/V4/(S(4hkhufsw5kohujphemn45ahu))/TripPinServiceRW/People

This returns a list of 8 people, 8 being a hard limit for the number of entities per result. In the real world, we would probably use a much larger limit. It does, however, provide an example of how OData includes additional hypertext links such as the @odata.nextLink, which we can use to fetch the next page of People in the search results.

We can also use query string values to narrow down the results list, such as selecting only the top 1 result.

https://services.odata.org/V4/(S(4hkhufsw5kohujphemn45ahu))/TripPinServiceRW/People?$top=1

We can also try filtering requests by FirstName.

https://services.odata.org/V4/(S(4hkhufsw5kohujphemn45ahu))/TripPinServiceRW/People?$filter=FirstName eq 'Russell'

In this instance, we used the eq operator to filter on all FirstNames that equal 'Russell'. Note the importance of wrapping strings in single quotes. OData provides a variety of different operators that can be used in combination to build up highly expressive search queries.

IRIS %Net Package

IRIS includes a comprehensive standard library. We ' ll be using the %Net package, which includes support for protocols such as FTP, Email, LDAP, and HTTP.

To use the TripPinWS service we will need to use HTTPS, which requires us to register an HTTPS configuration in the IRIS management portal. There are no complicated certificates to install so it ' s just a few steps:

- Open the IRIS management portal.
- Click on System Administration > Security > SSL/TLS Configurations.
- Click the "Create New Configuration" button.
- Enter the name "odataorg" and hit save.
- You can choose any name you ' d like, but we ' ll be using odataorg for the rest of the article.

We can now use the HttpRequest class to get a list of all people. If the Get() worked, then it will return 1 for OK. We can then access the response object and output the result to the terminal:

```
DC>set req=##class(%Net.HttpRequest).%New()
DC>set req.SSLConfiguration="odataorg"
DC>set sc=req.Get("https://services.odata.org:443/V4/(S(jndgbgy2tbu1vjtzyoei2w3e))/TripPinServiceRW/People")
DC>w sc
1
DC>do req.HttpResponse.OutputToDevice()
```

Feel free to experiment with the base HttpRequest before moving on. You could try fetching Airlines and Airports or investigate how errors are reported if you enter an incorrect URL.

Developing a generic OData Client

Let's create a generic OData client that will abstract the HttpRequest class and make it easier to implement various OData query options.

We ' ll call it DcLib.OData.Client and it will extend %RegisteredObject. We ' ll define several subclasses that we can

use to define the names of a specific OData service, as well as several properties that encapsulate runtime objects and values such as the HttpRequest object.

To make it easy to instantiate an OData client, we will also override the %OnNew() method (the class's constructor method) and use it to set up the runtime properties.

```
Class DcLib.OData.Client Extends %RegisteredObject
{

Parameter BaseURL;
Parameter SSLConfiguration;
Parameter EntityName;
Property HttpRequest As %Net.HttpRequest;
Property BaseURL As %String;
Property EntityName As %String;
Property Debug As %Boolean [ InitialExpression = 0 ];

Method %OnNew(pBaseURL As %String = "", pSSLConfiguration As %String = "") As %Status [ Private,
ServerOnly = 1 ]
{
  set ..HttpRequest=##class(%Net.HttpRequest).%New()
  set ..BaseURL=$select(pBaseURL'="":pBaseURL,1:..#BaseURL)
  set ..EntityName=..#EntityName
  set sslConfiguration=$select(pSSLConfiguration'="":pSSLConfiguration,1:..#SSLConfiguration)
  if sslConfiguration'="" set ..HttpRequest.SSLConfiguration=sslConfiguration
  quit $$$OK
}

}
```

We can now define a client class that is specific to the TripPinWS service by extending DcLib.OData.Client and setting the BaseURL and SSL Configuration parameters in one single place.

```
Class TripPinWS.Client Extends DcLib.OData.Client
{

Parameter BaseURL = "https://services.odata.org:443/V4/(S(jndgbgy2tbu1vjtzyoei2w3e))/TripPinServiceRW";
Parameter SSLConfiguration = "odataorg";

}
```

With this base client in place, we can now create a class for each entity type that we want to use in the service. By extending the new client class all we need to do is define the entity name in the EntityName parameter.

```
Class TripPinWS.People Extends TripPinWS.Client
{

Parameter EntityName = "People";

}
```

Next, we need to provide some more methods on the base DcLib.OData.Client class that will make it easy to query the entities.

```
Method Select(pSelect As %String) As DcLib.OData.Client
{
  do ..HttpRequest.SetParam("$select",pSelect)
  return $this
}
```

```
Method Filter(pFilter As %String) As DcLib.OData.Client
{
  do ..HttpRequest.SetParam("$filter",pFilter)
  return $this
}

Method Search(pSearch As %String) As DcLib.OData.Client
{
  do ..HttpRequest.SetParam("$search",pSearch)
  return $this
}

Method OrderBy(pOrderBy As %String) As DcLib.OData.Client
{
  do ..HttpRequest.SetParam("$orderby",pOrderBy)
  return $this
}

Method Top(pTop As %String) As DcLib.OData.Client
{
  do ..HttpRequest.SetParam("$top",pTop)
  return $this
}

Method Skip(pSkip As %String) As DcLib.OData.Client
{
  do ..HttpRequest.SetParam("$skip",pSkip)
  return $this
}

Method Fetch(pEntityId As %String = "") As DcLib.OData.ClientResponse
{
  if pEntityId="" return ##class(DcLib.OData.ClientResponse).%New($$$ERROR($$$GeneralError,"Entity ID must
be provided"),"")
  set pEntityId="('"_pEntityId_"')"
  if $extract(..BaseURL,*)'="/" set ..BaseURL=..BaseURL_"/"
  set sc=..HttpRequest.Get(..BaseURL_..EntityName_pEntityId,..Debug)
  set response=##class(DcLib.OData.ClientResponse).%New(sc,..HttpRequest.HttpResponse,"one")
  quit response
}

Method FetchCount() As DcLib.OData.ClientResponse
{
  if $extract(..BaseURL,*)'="/" set ..BaseURL=..BaseURL_"/"
  set sc=..HttpRequest.Get(..BaseURL_..EntityName_"/$count")
  set response=##class(DcLib.OData.ClientResponse).%New(sc,..HttpRequest.HttpResponse,"count")
  quit response
}

Method FetchAll() As DcLib.OData.ClientResponse
{
  #dim response As DcLib.OData.ClientResponse
  if $extract(..BaseURL,*)'="/" set ..BaseURL=..BaseURL_"/"
  set sc=..HttpRequest.Get(..BaseURL_..EntityName,..Debug)
  set response=##class(DcLib.OData.ClientResponse).%New(sc,..HttpRequest.HttpResponse,"many")
  if response.IsError() return response
  //if the response has a nextLink then we need to keep going back to fetch more data
  while response.Payload.%IsDefined("@odata.nextLink") {
  //stash the previous value array, push the new values on to it and then
```

```
  //set it back to the new response and create a new value iterator
  set previousValueArray=response.Payload.value
  set sc=..HttpRequest.Get(response.Payload."@odata.nextLink",..Debug)
  set response=##class(DcLib.OData.ClientResponse).%New(sc,..HttpRequest.HttpResponse)
  if response.IsError() return response
  while response.Value.%GetNext(.key,.value) {
  do previousValueArray.%Push(value)
  }
  set response.Payload.value=previousValueArray
  set response.Value=response.Payload.value.%GetIterator()
  }
  return response
}
```

We've added nine new methods. The first six are instance methods for defining query options, and the last three are methods for fetching one, all, or a count of all entities.

Notice that the first six methods are essentially a wrapper for setting parameters on the HTTP request object. To make implementation coding easier, each of these methods returns an instance of this object so that we can chain the methods together.

Before we explain the main Fetch() method let ' s see the Filter() method in action.

```
set people=##class(TripPinWS.People).%New().Filter("UserName eq 'ronaldmundy'").FetchAll()
while people.Value.%GetNext(.key,.person) {
 write !,person.FirstName," ",person.LastName
}
```

If we use this method, it returns:
Ronald Mundy

The example code creates an instance of the TripPinWS People object. This sets the base URL and certificate configuration in its base class. We can then call its Filter method to define a filter query and then FetchAll() to trigger the HTTP request.

Note that we can directly access the people results as a dynamic object, not as raw JSON data. This is because we are also going to implement a ClientResponse object that makes exception handling simpler. We also generate dynamic objects depending on the type of result that we get back.

First, let's discuss the FetchAll() method. At this stage, our implementation classes have defined the OData URL in its base class configuration, the helper methods are setting additional parameters, and the FetchAll() method needs to build the URL and make a GET request. Just as in our original command-line example, we call the Get() method on the HttpRequest class and create a ClientResponse from its results.

The method is complicated because the API only returns eight results at a time. We must handle this in our code and use the previous result's nextLink value to keep fetching the next page of results until there are no more pages. As we fetch each additional page, we store the previous results array and then push each new result on to it.

The Fetch(), FetchAll() and FetchCount() methods return an instance of a class called DcLib.OData.ClientResponse. Let's create that now to handle both exceptions and auto deserialize valid JSON responses.

```
Class DcLib.OData.ClientResponse Extends %RegisteredObject
{

Property InternalStatus As %Status [ Private ];
Property HttpResponse As %Net.HttpResponse;
Property Payload As %Library.DynamicObject;
```

```
Property Value;

Method %OnNew(pRequestStatus As %Status, pHttpResponse As %Net.HttpResponse, pValueMode As %String
= "") As %Status [ Private, ServerOnly = 1 ]
{
  //check for immediate HTTP error
  set ..InternalStatus = pRequestStatus
  set ..HttpResponse = pHttpResponse
  if $$$ISERR(pRequestStatus) {
  if $SYSTEM.Status.GetOneErrorText(pRequestStatus)["<READ>" set
..InternalStatus=$$$ERROR($$$GeneralError,"Could not get a response from HTTP server, server could be
uncontactable or server details are incorrect")
  return $$$OK
  }

  //if mode is count, then the response is not JSON, its just a numeric value
  //validate that it is a number and return all ok if true, else let it fall through
  //to pick up any errors that are presented as JSON
  if pValueMode="count" {
  set value=pHttpResponse.Data.Read(32000)
  if value?1.N {
  set ..Value=value
  return $$$OK
  }
  }

  //serialise JSON payload, catch any serialisation errors
  try {
  set ..Payload={}.%FromJSON(pHttpResponse.Data)
  } catch err {
  //check for HTTP status code error first
  if $e(pHttpResponse.StatusCode,1)'="2" {
  set ..InternalStatus = $$$ERROR($$$GeneralError,"Unexpected HTTP Status Code
"pHttpResponse.StatusCode)
  if pHttpResponse.Data.Size>0 return $$$OK
  }
  set ..InternalStatus=err.AsStatus()
  return $$$OK
  }

  //check payload for an OData error
  if ..Payload.%IsDefined("error") {
  do ..HttpResponse.Data.Rewind()
  set error=..HttpResponse.Data.Read(32000)
  set ..InternalStatus=$$$ERROR($$$GeneralError,..Payload.error.message)
  return $$$OK
  }

  //all ok, set the response value to match the required modes (many, one, count)
  if pValueMode="one" {
  set ..Value=..Payload
  } else {
  set iterator=..Payload.value.%GetIterator()
  set ..Value=iterator
  }

  return $$$OK
}
```

```
Method IsOK()
{
  return $$$ISOK(..InternalStatus)
}

Method IsError()
{
  return $$$ISERR(..InternalStatus)
}

Method GetStatus()
{
  return ..InternalStatus
}

Method GetStatusText()
{
  return $SYSTEM.Status.GetOneStatusText(..InternalStatus)
}

Method ThrowException()
{
  Throw ##class(%Exception.General).%New("OData Fetch
Exception","999",,$SYSTEM.Status.GetOneStatusText(..InternalStatus))
}

Method OutputToDevice()
{
  do ..HttpResponse.OutputToDevice()
}

}
```

Given an instance of the ClientResponse object, we can first test to see if there was an error. Errors can happen on several levels, so we want to return them in a single, easy-to-use solution.

```
set response=##class(TripPinWS.People).%New().Filter("UserName eq 'ronaldmundy'").FetchAll()
if response.IsError() write !,response.GetStatusText() quit
```

The IsOK() and IsError() methods check the object for errors. If an error occurred, we can call GetStatus() or GetStatusText() to access the error, or use ThrowException() to pass the error to an exception handler.

If there is no error, then the ClientResponse will assign the raw payload object to the response Payload property:

```
set ..Payload={}.%FromJSON(pHttpResponse.Data)
```

It will then set the response Value property to the main data array within the payload, either as a single instance or as an array iterator to traverse many results.

I've put all of this code together in a single project on GitHub https://github.com/SeanConnelly/IrisOData/blob/master/README.md which will make more sense when reviewed as a whole. All of the following examples are included in the source GitHub project.

Using the OData Client

There is just one more method we should understand on the base Client class: the With() method. If you don't want to create an instance of every entity, you can instead use the With() method with just one single client class. The With() method will establish a new client with the provided entity name:

```
ClassMethod With(pEntityName As %String) As DcLib.OData.Client
{
  set client=..%New()
  set client.EntityName=pEntityName
  return client
}
```

We can now use it to fetch all people using the base Client class:

```
/// Fetch all "People" using the base client class and .With("People")
ClassMethod TestGenericFetchAllUsingWithPeople()
{
  #dim response As DcLib.OData.ClientResponse
  set response=##class(TripPinWS.Client).With("People").FetchAll()

  if response.IsError() write !,response.GetStatusText() quit

  while response.Value.%GetNext(.key,.person) {
  write !,person.FirstName," ",person.LastName
  }
}
```

Or, using an entity per class approach:

```
/// Fetch all "People" using the People class
ClassMethod TestFetchAllPeople()
{
  #dim people As DcLib.OData.ClientResponse
  set people=##class(TripPinWS.People).%New().FetchAll()

  if people.IsError() write !,people.GetStatusText() quit

  while people.Value.%GetNext(.key,.person) {
  write !,person.FirstName," ",person.LastName
  }
}
```

As you can see, they ' re very similar. The correct choice depends on how important autocomplete is to you with concrete entities, and whether you want a concrete entity class to add more entity-specific methods.

```
DC>do ##class(TripPinWS.Tests).TestFetchAllPeople()

Russell Whyte
Scott Ketchum
Ronald Mundy
⋯ more people
```

Next, let's implement the same for Airlines:

```
/// Fetch all "Airlines"
ClassMethod TestFetchAllAirlines()
{
  #dim airlines As DcLib.OData.ClientResponse
  set airlines=##class(TripPinWS.Airlines).%New().FetchAll()

  if airlines.IsError() write !,airlines.GetStatusText() quit

  while airlines.Value.%GetNext(.key,.airline) {
  write !,airline.AirlineCode," ",airline.Name
```

```
  }
}
```

And from the command line ...
DC>do ##class(TripPinWS.Tests).TestFetchAllAirlines()

AA American Airlines
FM Shanghai Airline
⋯ more airlines

And now airports:

```
/// Fetch all "Airports"
ClassMethod TestFetchAllAirports()
{
  #dim airports As DcLib.OData.ClientResponse
  set airports=##class(TripPinWS.Airports).%New().FetchAll()

  if airports.IsError() write !,airports.GetStatusText() quit

  while airports.Value.%GetNext(.key,.airport) {
  write !,airport.IataCode," ",airport.Name
  }
}
```

And from the command line...
DC>do ##class(TripPinWS.Tests).TestFetchAllAirports()

SFO San Francisco International Airport
LAX Los Angeles International Airport
SHA Shanghai Hongqiao International Airport
⋯ more airports

So far we ' ve been using the FetchAll() method. We can also use the Fetch() method to fetch a single entity using the entity ' s primary key:

```
/// Fetch single "People" entity using the persons ID
ClassMethod TestFetchPersonWithID()
{
  #dim response As DcLib.OData.ClientResponse
  set response=##class(TripPinWS.People).%New().Fetch("russellwhyte")

  if response.IsError() write !,response.GetStatusText() quit

  //lets use the new formatter to pretty print to the output (latest version of IRIS only)
  set jsonFormatter = ##class(%JSON.Formatter).%New()
  do jsonFormatter.Format(response.Value)
}
```

In this instance, we are using the new JSON formatter class, which can take a dynamic array or object and output it to formatted JSON.
DC>do ##class(TripPinWS.Tests).TestFetchPersonWithID()

```
{
 "@odata.context":"http://services.odata.org/V4/
(S(jndgbgy2tbu1vjtzyoei2w3e))/TripPinServiceRW/$metadata#People/$entity",
```

"@odata.id":"http://services.odata.org/V4/
(S(jndgbgy2tbu1vjtzyoei2w3e))/TripPinServiceRW/People('russellwhyte')",
"@odata.etag":"W/\"08D720E1BB3333CF\"",
"@odata.editLink":"http://services.odata.org/V4/
(S(jndgbgy2tbu1vjtzyoei2w3e))/TripPinServiceRW/People('russellwhyte')",
"UserName":"russellwhyte",
"FirstName":"Russell",
"LastName":"Whyte",
"Emails":[
"Russell@example.com",
"Russell@contoso.com"
],
"AddressInfo":[
{
"Address":"187 Suffolk Ln.",
"City":{
"CountryRegion":"United States",
"Name":"Boise",
"Region":"ID"
}
}
],
"Gender":"Male",
"Concurrency":637014026176639951
}

Persisting OData

In the final few examples, we will demonstrate how the OData JSON could be deserialized into persistent objects using the new JSON adapter class. We will create three classes — Person, Address, and City — which will reflect the Person data structure in the OData metadata. We will use the %JSONIGNOREINVALIDFIELD set to 1 so that the additional OData properties such as @odata.context do not throw a deserialization error.

```
Class TripPinWS.Model.Person Extends (%Persistent, %JSON.Adaptor)
{

Parameter %JSONIGNOREINVALIDFIELD = 1;
Property UserName As %String;
Property FirstName As %String;
Property LastName As %String;
Property Emails As list Of %String;
Property Gender As %String;
Property Concurrency As %Integer;

Relationship AddressInfo As Address [ Cardinality = many, Inverse = Person ];

Index UserNameIndex On UserName [ IdKey, PrimaryKey, Unique ];

}


Class TripPinWS.Model.Address Extends (%Persistent, %JSON.Adaptor)
{
Property Address As %String;
Property City As TripPinWS.Model.City;
Relationship Person As Person [ Cardinality = one, Inverse = AddressInfo ];
}
```

```
Class TripPinWS.Model.City Extends (%Persistent, %JSON.Adaptor)
{
Property CountryRegion As %String;
Property Name As %String;
Property Region As %String;
}
```

Next, we will fetch Russel Whyte from the OData service, create a new instance of the Person model, then call the %JSONImport() method using the response value. This will populate the Person object, along with the Address and City details.

```
ClassMethod TestPersonModel()
{
  #dim response As DcLib.OData.ClientResponse
  set response=##class(TripPinWS.People).%New().Fetch("russellwhyte")

  if response.IsError() write !,response.GetStatusText() quit

  set person=##class(TripPinWS.Model.Person).%New()

  set sc=person.%JSONImport(response.Value)
  if $$$ISERR(sc) write !!,$SYSTEM.Status.GetOneErrorText(sc) return

  set sc=person.%Save()
  if $$$ISERR(sc) write !!,$SYSTEM.Status.GetOneErrorText(sc) return
}
```

We can then run a SQL command to see the data is persisted.

```
SELECT ID, Concurrency, Emails, FirstName, Gender, LastName, UserName
FROM TripPinWSModel.Person
```

ID Concurrency Emails FirstName Gender LastName UserName
russellwhyte 637012191599722031 Russell@example.com Russell@contoso.com Russell Male Whyte russellwhyte

Final Thoughts

As we ' ve seen, it ' s easy to consume RESTful OData services using the built-in %NET classes. With a small amount of additional helper code, we can simplify the construction of OData queries, unify error reporting, and automatically deserialize JSON into dynamic objects.

We can then create a new OData client just by providing its base URL and, if required, an HTTPS configuration. We then have the option to use this one class and the .With('entity') method to consume any entity on the service, or create named subclasses for the entities that we are interested in.

We have also demonstrated that it's possible to deserialize JSON responses directly into persistent classes using the new JSON adaptor. In the real world, we might consider denormalizing this data first and ensure that the JSON adapter class works with custom mappings.

Finally, working with OData has been a real breeze. The consistency of service implementation has required much less code than I often experience with bespoke implementations. Whilst I enjoy the freedom of RESTful design, I would certainly consider implementing a standard in my next server-side solution.

#REST API #Caché #InterSystems IRIS
Check the related application on InterSystems Open Exchange

Source URL:https://community.intersystems.com/post/creating-odata-api-adapter-intersystems-iris