記事
Tomohiro Iwamoto · 2020年11月23日 21m read

Open Exchange

InterSystemsIRIS 用 ODataAPI アダプタの作成

この記事では、OData API 標準に基づいて開発された RESTful API サービスを利用するための IRIS クライアントの開発について説明します。

HTTP リクエストを作成し、JSON ペイロードの読み取りと書き込みを行い、それらを組み合わせて OData 用の汎用クライアントアダプタを構築する方法を確認するため、多数の組み込み IRIS ライブラリを説明します。また、JSON を永続オブジェクトに逆シリアル化するための新しい JSON アダプタについても説明します。

RESTful API の操作

REST は World Wide Web の標準化に関する作業から作成された一連の設計原則です。これらの原則はあらゆるクライアントサーバー通信に適用でき、HTTP API が RESTful であることを説明するためによく使用されます。

REST はステートレスなリクエスト、キャッシュ処理、統一した API 設計など、さまざまな原則を網羅しています。
ただし、詳細な実装については網羅していません。また、これらのギャップを埋めるための一般的な API 仕様は存在しません。

この曖昧さは、RESTful API に幾分かの理解、ツール、より厳密なエコシステムを中心によく構築されるライブラリが不足している原因となっています。 特に、開発者は RESTful API の検出と文書化のために独自のソリューションを構築する必要があります。

OData

OData は、一貫性のある RESTful API を構築するための OASIS の仕様です。 OASIS コミュニティには、Microsoft / Citrix / IBM / Red Hat / SAP などの有名なソフトウェア会社が参加しています。2007 年に OData 1.0 が最初に導入され、最新バージョンの 4.1 が今年リリースされました。

OData の仕様は、メタデータ、一貫性のある操作の実装、クエリ、例外処理などを対象としています。また、アクションや関数などの追加機能も対象としています。

TripPinWS OData API の説明

この記事では、Odata.org が例示している TripPinWS API を使用します。

他の RESTful API と同様に、一般的にはサービスのベース URL が必要です。 OData でこのベース URL にアクセスすると、API エンティティのリストも返されます。

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

この API には Photos、People、Airlines、Airports、Me のエンティティと、GetNearestAirport という関数が含まれていることがわかります。

応答には、TripPinWS メタデータドキュメントへのリンクも含まれています。

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

このメタデータは XML ドキュメントとして実装されており、独自の XSD ドキュメントが含まれています。これにより、IRIS XML

スキーマウィザードから生成されるコードを使用してメタデータドキュメントを消費する可能性が広がります。

メタデータドキュメントは一見かなり複雑に見えるかもしれませんが、エンティティのスキーマ定義を構成するために使用されるタイプのプロパティを表しているだけです。

次の URL を使用すると、API から People のリストを取得できます。

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

この URL は 8 人のリストを返します。この 8 という数値は、厳密な結果ごとのエンティティ数の上限値です。実際には、これよりもはるかに大きな上限値を使用することになるでしょう。 ただし、OData が
@odata.nextLink
などの追加のハイパーテキストリンクを含んでいる例も示されています。そのリンクを使用すると、People を検索した結果の次のページを取得できます。

また、次のようにして上位 1
件の結果のみを選択するなど、クエリ文字列値を使用して結果リストを絞り込むこともできます。

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

FirstName でリクエストを絞り込むこともできます。

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

この例では eq 演算子を使用し、「Russell」に等しいすべての FirstName を抽出しました。
ここでは対象の文字列を一重引用符で囲むことが重要です。 OData
では、さまざまな演算子を組み合わせて表現力の高い検索クエリを作成することができます。

IRIS %Net パッケージ

IRIS には、包括的な標準ライブラリが含まれています。 私たちは FTP / メール / LDAP / HTTP などのプロトコルをサポートする %Net パッケージを使用することになります。

TripPinWS サービスを使用するには、HTTPS を使用する必要があります。そのためには、IRIS 管理ポータルに HTTPS 構成を登録する必要があります。
複雑な証明書をインストールする必要はないため、次のように数ステップで作業を完了することができます。

- IRIS 管理ポータルを開きます。
- [システム 管理] > [セキュリティ] > [SSL/TLS 構成] をクリックします。
- [新規構成の作成] ボタンをクリックします。
- 「odataorg」という構成名を入力し、[保存] をクリックします。
- ここでは任意の名前を選択できますが、記事の残りの部分では odataorg を使用します。

これで、HttpRequest クラスを使用して全員のリストを取得できるようになりました。 Get() が動作すると、OK の場合に 1 が返ってきます。 その後、次のように応答オブジェクトにアクセスして結果を端末に出力できます。

```
 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()
```

先に進む前に、基本的な HttpRequest を自由に試してみてください。 Airlines や Airports を取得してみたり、不正な URL を入力した場合にどのようなエラーが返ってくるかを調べたりしてください。

汎用 OData クライアントの開発

HttpRequest クラスを抽象化し、さまざまな OData クエリオプションの実装を簡単にする汎用 OData クライアントを作成しましょう。

このクライアントを DcLib.OData.Client と名付け、%RegisteredObject を拡張して作成します。 また、特定の OData サービスの名前を定義するために使用できるいくつかのサブクラスと、HttpRequest オブジェクトなどのランタイムオブジェクトと値をカプセル化するいくつかのプロパティを定義します。

OData クライアントのインスタンス化を簡単にするため、%OnNew() メソッド（クラスのコンストラクタメソッド）もオーバーライドし、それを使用して実行時のプロパティを設定します。

```
  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
}

}
```

このように DcLib.OData.Client を拡張し、BaseURL と SSL 構成パラメーターを一箇所で設定することにより、TripPinWS サービスに固有のクライアントクラスを定義できるようになります。

```
  Class TripPinWS.Client Extends DcLib.OData.Client
{
Parameter BaseURL = "https://services.odata.org:443/V4/(S(jndgbgy2tbu1vjtzyoei2w3e))/TripPinServiceRW";
Parameter SSLConfiguration = "odataorg";

}
```

この基本クライアントを使用すれば、サービスで使用したいエンティティタイプごとにクラスを作成できます。この新しいクライアントクラスを拡張すれば、EntityName パラメーターでエンティティ名を定義するだけで済みます。

```
  Class TripPinWS.People Extends TripPinWS.Client
{
Parameter EntityName = "People";

}
```

次に、エンティティのクエリを簡単にするため、基本の DcLib.OData.Client クラスにさらにいくつかのメソッドを追加する必要があります。

```
  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
  //応答に nextLink が含まれる場合は、さらにデータを取得し続ける必要があります。
```

```
  while response.Payload.%IsDefined("@odata.nextLink") {
  //前の値を配列に退避し、新しい値をその配列にプッシュしてから
  //新しい応答にそれを設定し直し、新しい値のイテレータを作成します。
  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
}
```

ここでは 9 つの新しいメソッドを追加しました。 最初の 6
つはクエリオプションを定義するためのインスタンスメソッドであり、最後の 3 つは 1 つのエンティティ、すべてのエンティティ、またはすべてのエンティティのカウントを取得するためのメソッドです。

最初の 6 つのメソッドは、基本的に HTTP
リクエストオブジェクトにパラメーターを設定するためのラッパーです。 コードを実装しやすくするため、これらの各メソッドはこのオブジェクトのインスタンスを返し、メソッドをチェーン化できるようにしています。

メインの Fetch() メソッドについて説明する前に、Filter() メソッドの動作を見てみましょう。

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

このメソッドを使用すると、次のような結果が返ってきます。

```
  Ronald Mundy
```

このサンプルコードでは、TripPinWS Peopleオブジェクトのインスタンスを作成しています。
これにより、ベース URL と基本クラスの証明書の構成が設定されます。 その後、その Filter
メソッドを呼び出してフィルタークエリを定義してから FetchAll() を呼び出すと HTTP
リクエストを呼び出すことができます。

ここで直接アクセスできるのは生の JSON データではなく、動的オブジェクトとしての People
の結果であることに注意してください。 これは、例外処理を簡単にする ClientResponse
オブジェクトも実装するためです。 また、返される結果のタイプに応じて動的オブジェクトを生成します。

まず、FetchAll() メソッドについて説明します。 この段階で、実装クラスが基本クラス構成で OData URL
を定義しており、ヘルパーメソッドが追加のパラメーターを設定しています。FetchAll() メソッドは URL
を組み立てて GET リクエストを発行する必要があります。 元のコマンドラインの例と同様に HttpRequest
クラスで Get() メソッドを呼び出し、その結果から ClientResponse を作成します。

API が一度に 8 件しか結果を返さないため、このメソッドは複雑になっています。
コード内でこの制限に対処し、前の結果の nextLink
値を使用して最後のページに到達するまで次の結果ページを取得し続ける必要があります。 追加の各ページを取得する際に前の結果配列を保管してから、新しい結果をそれぞれその配列にプッシュしています。

Fetch() / FetchAll() / FetchCount() メソッドは、DcLib.OData.ClientResponse
というクラスのインスタンスを返します。 例外を処理し、有効な JSON
応答を自動的に逆シリアル化するためにこのクラスを作成しましょう。

```
  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 ]
{
  //直接の HTTP エラーをチェック
  set ..InternalStatus = pRequestStatus
  set ..HttpResponse = pHttpResponse
  if $$$ISERR(pRequestStatus) {
  if $SYSTEM.Status.GetOneErrorText(pRequestStatus)["" set
..InternalStatus=$$$ERROR($$$GeneralError,"Could not get a response from HTTP server, server could be
uncontactable or server details are incorrect")
  return $$$OK
  }

  //モードが count の場合、応答は JSON ではなく単なる数値になります。
  //数値であることを確認し、true ならばすべて ok を返しますが、それ以外の場合は
  //JSON で表現されるエラーを検出するためにフォールスルーします。
  if pValueMode="count" {
  set value=pHttpResponse.Data.Read(32000)
  if value?1.N {
  set ..Value=value
  return $$$OK
  }
  }

  //JSON ペイロードをシリアル化し、シリアル化エラーをキャッチします。
  try {
  set ..Payload={}.%FromJSON(pHttpResponse.Data)
  } catch err {
  //先に HTTP ステータスコードのエラーをチェックします。
  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
  }

  //OData エラーのペイロードをチェックします。
  if ..Payload.%IsDefined("error") {
  do ..HttpResponse.Data.Rewind()
  set error=..HttpResponse.Data.Read(32000)
  set ..InternalStatus=$$$ERROR($$$GeneralError,..Payload.error.message)
  return $$$OK
  }

  //すべて ok なら、必要なモード（many, one, count）に一致するように応答値を設定します。
  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()
}

}
```

ClientResponse オブジェクトのインスタンスが与えられた場合、最初にテストを実行してエラーがあったかどうかを確認することができます。 エラーは複数のレベルで発生する可能性があるため、単一の使いやすいソリューションでエラーを返すのが望ましいです。

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

IsOK() メソッドと IsError() メソッドはオブジェクトのエラーをチェックします。 エラーが発生した場合は GetStatus() または GetStatusText() を呼び出してエラーにアクセスするか、ThrowException() を使用してエラーを例外ハンドラに渡すことができます。

エラーが発生していない場合、ClientResponse は生のペイロードオブジェクトを応答ペイロードのプロパティに代入します。

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

次に、応答の Value プロパティを単一のインスタンスとして、または多数の結果を探索するための配列イテレータとして、ペイロード内のメインデータ配列に設定します。

私はこれらすべてのコードを GitHub https://github.com/SeanConnelly/IrisOData/blob/master/README.md 上の単一のプロジェクトに格納しています。そこで全体を見直せば、より深く理解することができるでしょう。次の例はすべて、ソースの GitHub プロジェクトに含まれています。

OData クライアントの使用

基本 Client クラスに関しては、With() メソッドも理解しておく必要があります。
すべてのエンティティのインスタンスを作成する代わりに、単一のクライアントクラスだけで With()
メソッドを使用することができます。 With()
メソッドは、指定されたエンティティ名で新しいクライアントを定義します。

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

このメソッドを使用すれば、次のように基本 Client クラスですべての People を取得できます。

```
 /// 基本クライアントクラスと .With("People") を使用してすべての "People" を取得します
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
 }
}
```

または、次のようにクラスごとのエンティティを使用します。

```
 /// People クラスを使用してすべての "People" を取得します
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
 }
}
```

ご覧のとおり、これらの方法は非常に似通っています。 どちらの方法を選択すべきかは、具体的なエンティティについて自動補完がどれほど重要であるか、および具体的なエンティティクラスにエンティティ固有のメソッドを追加するかどうかによって異なります。

```
 DC>do ##class(TripPinWS.Tests).TestFetchAllPeople()
Russell Whyte
Scott Ketchum
Ronald Mundy
… およびその他の人
```

次に、Airlines についても同じ処理を実装しましょう。

```
 /// すべての "Airlines" を取得します
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
  }
}
```

そして、コマンドラインから次の結果を得ることができます。

```
  DC>do ##class(TripPinWS.Tests).TestFetchAllAirlines()
AA American Airlines
FM Shanghai Airline
```

… およびその他の航空会社

次は Airports の実装です。

```
  /// すべての "Airports" を取得します
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
  }
}
```

そして、コマンドラインから次の結果を得ることができます。

```
  DC>do ##class(TripPinWS.Tests).TestFetchAllAirports()
SFO San Francisco International Airport
LAX Los Angeles International Airport
SHA Shanghai Hongqiao International Airport
```

… およびその他の空港

これまでは FetchAll() メソッドを使用してきました。 次のように Fetch()
メソッドを使用し、エンティティの主キーを使用して単一のエンティティを取得することもできます。

```
  /// 人の識別子を使用して単一の "People" エンティティを取得します
ClassMethod TestFetchPersonWithID()
{
  #dim response As DcLib.OData.ClientResponse
  set response=##class(TripPinWS.People).%New().Fetch("russellwhyte")

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

  //新しいフォーマッターを使用して出力を美しく整形してみましょう（最新バージョンの IRIS のみ）
  set jsonFormatter = ##class(%JSON.Formatter).%New()
  do jsonFormatter.Format(response.Value)
}
```

この例では動的配列またはオブジェクトを取得し、整形したJSONに出力できる新しい JSON
フォーマッタークラスを使用しています。

```
  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
}
```

OData の永続化

最後のいくつかの例では、新しい JSON アダプタクラスを使用して OData JSON を永続オブジェクトに逆シリアル化する方法を説明します。 ここでは Person、Address、City の 3 つのクラスを作成しますが、いずれも Person のデータ構造を OData メタデータに反映します。 また、@odata.context のような追加の OData プロパティが逆シリアル化エラーをスローしないよう、1 に設定された %JSONIGNOREINVALIDFIELD を使用します。

```
  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;
}
```

次に、OData サービスから Russel Whyte を取得し、Person

モデルの新しいインスタンスを作成した後に応答値を使用して %JSONImport() メソッドを呼び出します。
これにより、Address と City の詳細とともに Person オブジェクトにデータが入力されます。

```
 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
}
```

次に、次のように SQL コマンドを実行してデータが永続化されていることを確認できます。

```
 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
```

最終的な考え

上記のように、組み込みの %NET クラスを使用して RESTful な OData サービスを利用するのは簡単です。
少しばかりの追加ヘルパーコードを使用すれば、OData
クエリの構築を単純化し、エラーレポートを統合し、JSON
を動的オブジェクトに自動的に逆シリアル化できます。

そして、必要に応じてベース URL と HTTPS 構成を指定するだけで、新しい OData
クライアントを作成できます。 さらに、この単一のクラスと .With('エンティティ') メソッドを使用してサービス上の任意のエンティティを利用するか、関心のあるエンティティの名前付きサブクラスを作成することができます。

また、新しい JSON アダプタを使用して JSON
応答を永続クラスに直接逆シリアル化できることも説明しました。
現実的には最初にこのデータを非正規化することを検討し、JSON
アダプタクラスがカスタムマッピングで機能することを確認する必要があります。

最後になりますが、OData の操作は非常に簡単です。 私が特注の実装でよく経験する場合よりもはるかに少ないコード量でサービス実装の一貫性を維持することができました。 私は RESTful 設計の自由さを楽しんでいますが、次のサーバーサイドソリューションでは標準を実装することを検討したいと思います。

#REST API #Caché #InterSystems IRIS
InterSystems Open Exchangeで関連アプリケーションを確認してください

ソースURL:
https://jp.community.intersystems.com/post/intersystemsiris-%E7%94%A8-odataapi-%E3%82%A2%E3%83%80%E3%83%97%E3%82%BF%E3%81%AE%E4%BD%9C%E6%88%90