Article
Matthew Giesmann · Nov 20, 2020 6m read

Leveraging Queries and ObjectScript with the AppS.REST Framework

Earlier this year, the AppS.REST package was released. AppS.REST is a framework for easily exposing existing persistent classes in IRIS as REST resources. AppS.REST-enabled classes support CRUD operations with little effort from the developer, bridging the gap between persistent data in IRIS and data consumers, such as an Angular front end application.

But IRIS classes are much more than just a definition for loading and saving individual records! This article aims to highlight a few ways to leverage the power of IRIS in your REST applications. Using the Phone.Contact sample app, we'll look at out-of-the-box query support, use of class queries and finally ObjectScript methods.

Getting Started

You can find the AppS.REST package on the Open Exchange. This article will use examples from the Sample Phonebook app (available on github). Both packages can be installed easily with the Objectscript Package Manager, which can be found here.

Once installed, the Phonebook sample app will populate some dummy data and set up the necessary web application. That web application will forward all requests to the Sample.Phonebook.REST.Handler class. I'm testing locally, so all of my http requests will be under http://localhost:52773/csp/USER/phonebook-sample/api/, but your server and port my differ.

I used the Talend API Tester Chrome extension to try out all of these REST calls, but there are many good API tools out there.

Enabling Simple Requests

Let's take a look at the Model.Person class, which is enabled for REST by extending AppS.REST.Model.Adaptor. We see that it is exposed as the "contact" resource as defined by the RESOURCENAME parameter.

```
Class Sample.Phonebook.Model.Person Extends (%Persistent, %Populate, %JSON.Adaptor, A
ppS.REST.Model.Adaptor)
{
Parameter RESOURCENAME = "contact";
```

Note that even though the IRIS class is called "Person", we are exposing the class as the "contact" resource to our REST consumers. Let's start off with a basic GET request to demonstrate how a data consumer can interact with the contact resource:

http://localhost:52773/csp/USER/phonebook-sample/api/contact/2

```
{
"_id": "2",
"name": "Harrison,Angela C.",
"phones":[
```

```
{"_id": "2||15","number": "499-388-2049","type": "Office"},
{"_id": "2||32","number": "227-915-3954","type": "Mobile"}
]}
```


Great, we're already consuming IRIS data over REST!

Out-of-the-Box Queries

The above is good for loading a particular known instance of a resource, but what if you want to leverage IRIS's querying capabilities? The AppS.REST framework gives us some querying abilities right out of the box. For example the following GET request fetches all contacts:

http://localhost:52773/csp/USER/phonebook-sample/api/contact

Notice, it looks the same as above, only without a contact id specified. To request contacts with a given name, we can add a URL parameter:

http://localhost:52773/csp/USER/phonebook-sample/api/contact?name[eq]=Harrison,Angela C.


```
[{
"_id": "2",
"name": "Harrison,Angela C.",
"phones":[{"_id": "2||15", "number": "499-388-2049", "type": "Office"…]
}]
```


We can imagine an app that allows the user to search contacts by the first letters in their last name. We can achieve that with another operator:

http://localhost:52773/csp/USER/phonebook-sample/api/contact?name[stwith]=Harri

We now get back all contacts whose names start with "Harri":


```
[{
"_id": "2",
"name": "Harrison,Angela C.",
"phones":[{"_id": "2||15", "number": "499-388-2049", "type": "Office"…]
},{
"_id": "47",
"name": "Harrison,Yan N.",
"phones":[{"_id": "47||26", "number": "372-757-5547", "type": "Mobile" },…]
}]
```


AppS.REST supports 7 operators that translate directly into SQL:

```
"lte": " < "
"gte": " > "
"eq": " = "
"leq": " <= "
"geq": " >= "
"stwith": " %startswith "
"isnull": " is null"
```

Exposing Class Queries As REST Actions

What if we want to leverage more complicated or previously existing queries in our REST application? We can expose any class query as a REST Action!

The Person class has a query called FindByPhone, which uses the `PhoneNumber` table to look up People by a phoneFragment:

```
Query FindByPhone(phoneFragment As %String) As %SQLQuery
{
select distinct Person
from Sample_Phonebook_Model.PhoneNumber
where $Translate(PhoneNumber,' -+()') [ $Translate(:phoneFragment,' -+()')
}
```

The AppS.REST framework will automatically generate the appropriate JSON representation from the ids returned by the query.

To expose the query, we add an entry in Person's ActionMap XData block to define a "find-by-phone" endpoint under the contact resource:

```
 XData ActionMap
<action name="find-by-phone" target="class" method="GET" query="FindByPhone">
<argument name="phoneFragment" target="phoneFragment" source="url" />
</action>
```

That action now targets the FindByPhone class query, and it expects one argument for phoneFragment as a URL parameter.

Now if we make a GET request to the contact resource's $find-by-phone endpoint, we get back all contacts that have a matching phone number:
http://localhost:52773/csp/USER/phonebook-sample/api/contact/$find-by-phone?phoneFragment=641


```
[{
"_id": "19",
"name": "Ipsen,Rob Z.",
"phones":[{"_id": "19||211", "number": "641-489-2449", "type": "Home"}]
},{
"_id": "86",
"name": "Newton,Phil Y.",
"phones":[{"_id": "86||108", "number": "380-846-4132", "type": "Mobile"}]
}]
```


Exposing ObjectScript Methods As REST Actions

Exposing queries can be useful, but what if we want to expose application code written in ObjectScript? Let's take a look at Model.Person's AddPhoneNumber method:

```
Method AddPhoneNumber(phoneNumber As Sample.Phonebook.Model.PhoneNumber) As Sample.Ph
onebook.Model.Person
{
    Set phoneNumber.Person = $This
    $$$ThrowOnError(phoneNumber.%Save())
    Quit $This
}
```

This instance method is called to add a phone number to an existing person, so lets see how it can be exposed as

a REST action, much like the class query example above.

In Model.Person's ActionMap, we see another entry that calls the AddByPhone Method:

```
<action name="add-phone" target="instance" method="POST" call="AddPhoneNumber">
<argument name="phoneNumber" target="phoneNumber" source="body" />
</action>
```

Unlike the find-by-phone action, this one targets an instance of Model.Person, so our request URL will need to include an id after the resource:

http://localhost:52773/csp/USER/phonebook-sample/api/contact/2/$add-phone

The AppS.REST framework will automatically instantiate an object for the Person with id=2, and run the instance method on it.

The action map defines this as a POST action, and it expects a phoneNumber in the body of the request. AppS.REST will automatically translate the json body of the post into an instance of the Phone class since Phone also extends AppS.REST.Model.Adaptor.

Putting it together, we POST to http://localhost:52773/csp/USER/phonebook-sample/api/contact/2/$add-phone with a body:

```
{"number":"123-456-7890","type":"Mobile"}
```

If the request succeeds, the phone number is added to our contact and we receive back the full Person object, new phone number included:

```
{
"_id": "2",
"name": "Harrison,Angela C.",
"phones":[
{"_id": "2||301", "number": "123-456-7890", "type": "Mobile"},
{"_id": "2||15", "number": "499-388-2049", "type": "Office"},
{"_id": "2||32", "number": "227-915-3954", "type": "Mobile"}
]}
```

Actions are very flexible, allowing you to expose just about any class fuctionality to your REST consumers.

Wrapping Up

I hope this quick exploration of the Sample.Phonebook app has pointed you toward the many mays the AppS.REST framework makes quickly exposing IRIS classes over REST easy. AppS.REST.Model.Adaptor allows you to enable CRUD operations over REST for your IRIS classes with very few manual steps. Exposing class queries and ObjectScript methods can be done by defining Actions in your class's ActionMap.

#REST API #InterSystems IRIS