
Article

[Rubens Silva](#) · Aug 9, 2017 9m read

[Open Exchange](#)

Frontier: An abstraction layer for rapid REST development - Part 1 - Core concepts

Hello.

The idea of this post is to introduce Frontier: An abstraction layer that allows Rapid REST development.

REQUIREMENTS:

- Caché 2016.2 or higher.
- [Frontier](#).

Why?

Have you ever found yourself dealing with repetitive tasks like mounting objects, serializing them and eventually handling multiple errors for multiple cases? Frontier can boost your development by making you focus on what really matters: your application.

Frontier is made to stop you from WRITE'ing by instead forcing your methods to return values.

It's designed to make you code clean, and you'll see the why pretty soon.

This is the Part 1, where you'll learn the basics about how to work with Frontier. That means at the end of this part you should be capable of

creating GET requests without difficulties. Since this also serves

as a way to introduce the framework, I'll be calling this part:
Core concepts.

1. Core concepts

- Getting started
- Creating a simple request
- Query parameters
- Aliasing query parameters
- Changing output format
- Rest query parameters
- Inferring object instances
- Using literal notation
- Seamlessly mixing instances with literals
- Returning streams

2. [Handling payloads](#)

- How it works
- Making it useful
- Unmarshalling payloads into instances
- Using the unmarshaller to EDIT an existing object

3. [Using the SQL API](#)

- Creating a simple dynamic query
- Overwriting the default container property
- Using cached queries
- Passing parameters to queries

4. [Sharing data across router methods](#)

5. Forcing API errors

6. Managing errors with Reporters

- What are reporters
- Basic setup
- Using error filters

7. Authentication

- What are strategies
- Basic setup
- Disabling authentication for a specific route
- Enforcing a strategy usage for a specific route

Getting started

Before anything else you need to setup a new web application. For this demonstration I'll be using my local machine, but you're free to choose where you want to host the API.

Create a new web application, this is the same procedure you would do when creating a dispatch class for a %CSP.REST based application.

Use the following form to create a new web application:

Name	<input type="text" value="/api/frontier/demo"/> <small>Required. (e.g. /csp/appname)</small>		
Copy from	<input type="text"/>		
Description	<input type="text"/>		
Namespace	<input type="text" value="DEV"/>	<input type="checkbox"/> Namespace Default Application	
Enabled	<input checked="" type="checkbox"/> Application <input checked="" type="checkbox"/> CSP/ZEN <input checked="" type="checkbox"/> Inbound Web Services <input type="checkbox"/> DeepSee <input type="checkbox"/> iKnow		
Permitted Classes	<input type="text"/>		
Security Settings	<div>Resource Required <input type="text"/> Group By ID <input type="text"/></div> <div>Allowed Authentication Methods <input checked="" type="checkbox"/> Unauthenticated <input type="checkbox"/> Password <input type="checkbox"/> Login Cookie</div>		
Session Settings	<div>Session Timeout <input type="text" value="900"/> seconds Event Class <input type="text"/></div> <div>Use Cookie for Session <input type="text" value="Always"/> Session Cookie Path <input type="text" value="/api/frontier/demo/"/></div>		
Dispatch Class	<input type="text" value="Frontier.Demo"/>		
CSP File Settings	<div>Serve Files <input type="text" value="Always"/> Serve Files Timeout <input type="text" value="3600"/> seconds</div> <div>CSP Files Physical Path <input type="text"/> <input type="button" value="Browse..."/></div> <div>Package Name <input type="text"/> Default Superclass <input type="text"/></div> <div>CSP Settings <input checked="" type="checkbox"/> Recurse <input checked="" type="checkbox"/> Auto Compile <input checked="" type="checkbox"/> Lock CSP Name</div>		
Custom Pages	<div>Login Page <input type="text"/> Change Password Page <input type="text"/></div> <div>Custom Error Page <input type="text"/></div>		

This is where things became different, you'll notice that instead of using %CSP.REST, we're going to use Frontier.Router instead.

Creating a simple request

Create a new class that extends from Frontier.Router instead of %CSP.REST, and define a Route like the code below:

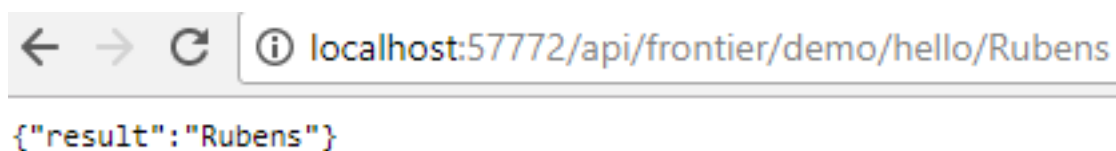
Class Frontier.Demo Extends Frontier.Router

```
{  
  
XData UriMap [ XMLNamespace = "https://github.com/rfns/frontier" ]  
{  
<Routes>  
  <Route Url="/hello/:name" Method="GET" Call="SayHello"></Route>  
</Routes>  
}  
}
```

Now add the method like below:

```
ClassMethod SayHello(name As %String) As %String  
{  
  return name  
}
```

And use a browser of your preference to see the result. And you'll notice something like this:



This is sufficient to mimic most of %CSP.REST's output but with a single line.
Notice that instead of write'ing we simply made the method return a value.

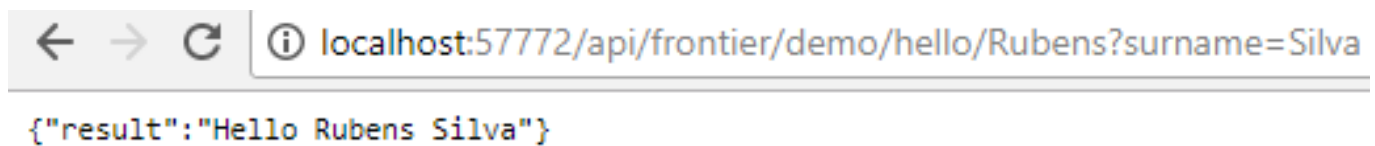
Query parameters

Now that you know how to do a simple request, let's go further and demonstrate how it's possible to define and use query parameters.

First, we modify the method SayHello to look like this:

```
ClassMethod SayHello(name As %String, surname As %String = "") As %String
{
    return $$$FormatText("%1 %2", name, surname)
}
```

Now back to the browser, so that we modify the url to call the new method.



Notice that the query parameter is matching our method's argument name. This is the default behavior.

We'll see how we can make a query parameter with a different name but still matching our argument.

Aliasing query parameters

There can be cases where you're actually migrating from another back-end, normally this would also imply on updating query parameters, which is a tiresome task.

So instead why not make the router compatible? As you should have noticed for now, Frontier uses argument names binded by their query parameters counterpart.

This means, if you have a situation where a query parameter is not support by a method argument syntax, like "personid", you're out of luck.

Or, that would be the case. If not for the alias support.

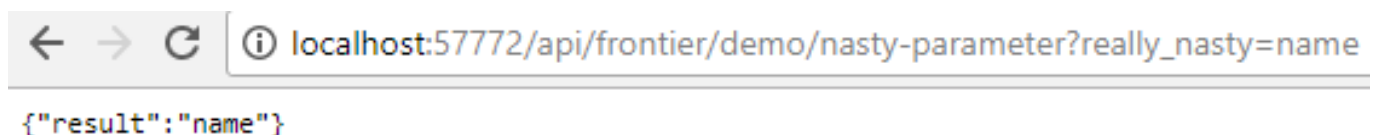
Add a new route like this:

```
<Route Url="/nasty-parameter" Method="GET" Call="DealWithThisNastyParameter"></Route>
```

And the method to deal with this nasty parameter!

```
ClassMethod DealWithThisNastyParameter(notSoNastyNowHuh As %String(ALIAS="really_nasty" )) As %String
{
    return notSoNastyNowHuh
}
```

Now you should be able to use your legacy client-side application without changing its request.



Changing output format

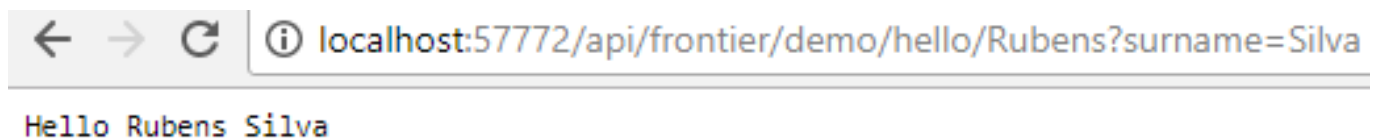
By now, you should have noticed that we only worked with JSON so far.

If you want to work with something other than JSON, simply call the Raw method.

Now take a note on that %frontier variable, it's an object representing the context of the request.

```
ClassMethod SayHello(  
    name As %String,  
    surname As %String = "") As %String  
{  
    do %frontier.Raw()  
    return $$$FormatText("Hello %1 %2", name, surname)  
}
```

And voilà! The JSON format is gone as now we are only seeing the resulting string.



For this tutorial we're going to work with JSON only, but you can also change the header using HTML(), turn back to JSON using JSON(), check which format is selected using IsJSON, IsHTML and so on. Well, there isn't much difference between HTML and Raw for now, except that %response.ContentType is modified accordingly, but it's good to know.

Moving back to JSON, remove the line we added and the content should be written using JSON format again.

Rest query parameters

This name can be a bit misleading, since we've been handling the REST expression all this time. But for this case, rest (notice the lowercase), means an argument that can receive a variable arity and are normally identified by three dots.

That being said, requests can also have a dynamic amount of a specific query parameters with the same name, this is required to match a rest argument syntax.

Frontier is capable of handling this type of request as well.

Let's see how that works, add the following route to your UriMap:

```
<Route Url="/sum" Method="GET" Call="SumTheseValues"></Route>
```

And define the following method:

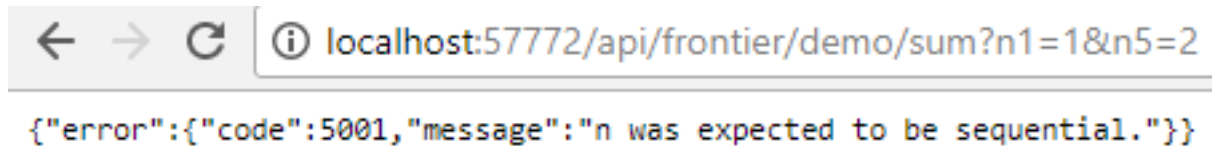
```
ClassMethod SumTheseValues(n... As %Integer) As %Integer  
{  
    set sum = 0  
    for i=1:1:n set sum = sum + n(i)  
    return sum  
}
```

Notice that 'n', it's the query parameter name that the request must pass to execute as we want. We expected the

request to send n1, n2 and so on. Back to our browser, we get the following result:

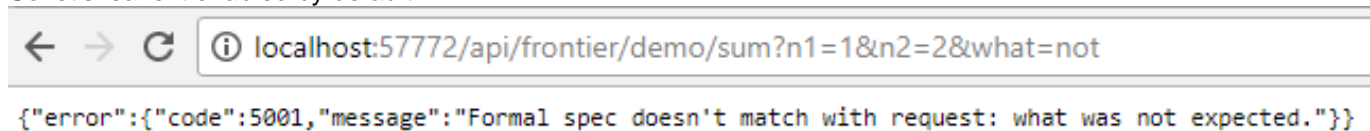


Naturally, the request must follow the sequence otherwise an error is thrown:



And there's also a validation for unknown parameters too, even though that is configurable. For now, it's out of this scope.

So let's leave it enabled by default.



Inferring object instances

Inferring an object instance means using a data value to open it whose type is deduced from the typed arguments.

Frontier can auto-open these instances using both values: from route parameters or query parameters.

If no valid value is provided, the argument is set to empty, it's the developer's responsibility to handle this situation.

To understand how Frontier does inferring, add the following route:

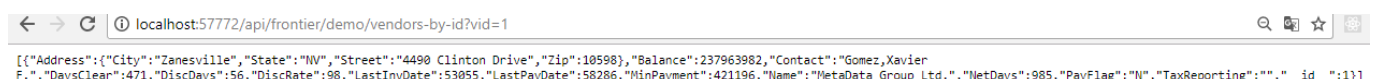
```
<Route Url="/vendors-by-id" Method="GET" Call="GetVendorsById"></Route>
```

So now we rewrite our method to look like this:

```
ClassMethod GetVendorsById(vid... As Sample.Vendor) As %DynamicArray
{
    set vendors = []
    for i=1:1:vid do vendors.%Push(vid(i))
    return vendors
}
```

Notice the bold part, this is where we signal Frontier about which type should the instance be inferred from. This example also shows how it's possible to mix both features

for a more resourceful result.



Using literal notation

Check out the usage of `[]` it's the literal format for creating a `%DynamicArray` instance.

Just like JavaScript, the newest Caché versions make the process of creating such instances easier.

- `{}` creates an empty `%DynamicObject` instance.
- `[]` creates an empty `%DynamicArray` instance.

We are going to see how each works and even use it with existing instances. So let's get started with an example using the `%DynamicObject`.

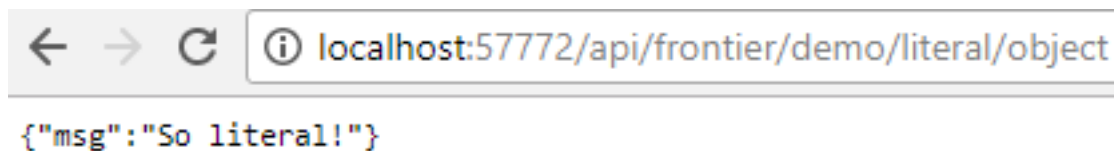
Add a new route:

```
<Route Url="/literal/object" Method="GET" Call="GetLiteralObject"></Route>
```

And now a new method to bind to it.

```
ClassMethod GetLiteralObject() As %DynamicObject
{
    return { "msg": "So literal!" }
}
```

Now navigate the this route and you should notice something different:



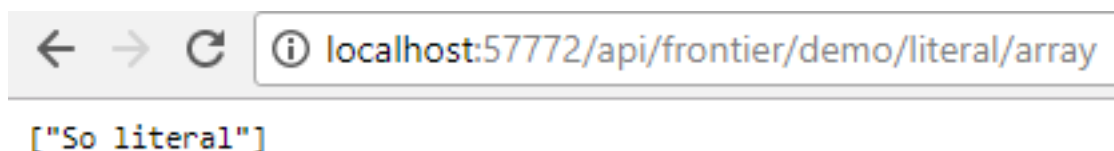
Did you find it? Yes! The "result" key is gone for good! The point here is that whenever a `%Dynamic` instance is returned,

Frontier assumes that you wish to keep their original format instead of putting some undesirable property.

Now using an array, you can also notice the same behavior with `%DynamicArray`:

```
<Route Url="/literal/array" Method="GET" Call="GetLiteralArray"></Route>
ClassMethod GetLiteralArray() As %DynamicArray
{
    return [ "So literal" ]
}
```

And here's the result.



Even though the developer is able to return an array directly, smart ones wouldn't as this would [compromise](#) security to some level.

So here's the tip: **DON'T return an array, unless you really need to!**

Seamlessly mixing instances with literals

You can also mix both non-`%Dynamic` instances with them and make some feedback more flexible.

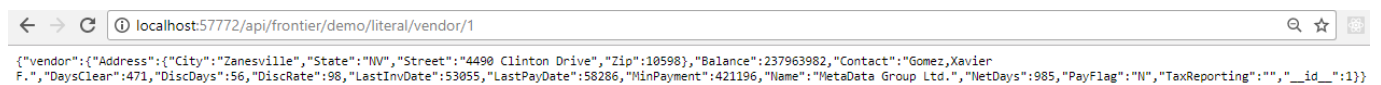
Add a new route like this:

```
<Route Url="/literal/vendor/:vendor" Method="GET" Call="GetVendorByRouteId"></Route>
```

And the method for it:

```
ClassMethod GetVendorByRouteId(vendor As Sample.Vendor) As %DynamicObject
{
    return {
        "vendor": (vendor)
    }
}
```

Now navigate to this route and the result should be something like this:



localhost:57772/api/frontier/demo/literal/vendor/1

```
{
  "vendor": {
    "Address": {
      "City": "Zanesville",
      "State": "WV",
      "Street": "4490 Clinton Drive",
      "Zip": "10598"
    },
    "Balance": 237963982,
    "Contact": "Gomez, Xavier F.",
    "DaysClear": 471,
    "DiscDays": 56,
    "DiscRate": 98,
    "LastInvDate": 53055,
    "LastPayDate": 58286,
    "MinPayment": 421196,
    "Name": "MetaData Group Ltd.",
    "NetDays": 985,
    "PayFlag": "N",
    "TaxReporting": "",
    "__id__": 1
  }
}
```

Notice the "vendor" property, this is exactly what we expected. By wrapping a Sample.Vendor instance inside our literal object, we could make it serialized inside without much effort.

Also, by saying "wrapping a Sample.Vendor " you can assume that the vendor has been inferred.

Returning streams

Big texts can use %Stream instances instead of strings, and this is obviously covered by Frontier as well. To see it working

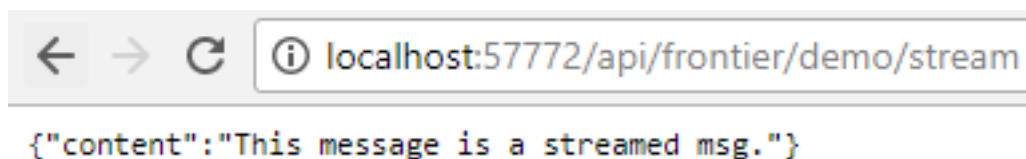
create a new route to handle this type of object:

```
<Route Url="/stream" Method="GET" Call="GetText"></Route>
```

And the method itself:

```
ClassMethod GetText() As %Stream.GlobalCharacter
{
    set stream = ##class(%Stream.GlobalCharacter).%New()
    do stream.Write("This message is a streamed msg.")
    return stream
}
```

And the result should be:



localhost:57772/api/frontier/demo/stream

```
{
  "content": "This message is a streamed msg."
}
```

Returning streams directly makes Frontier create an implicit %DynamicObject containing a property called "content".

Just like the "result", if you want to overwrite this behavior you need to return your custom %Dynamic instance.

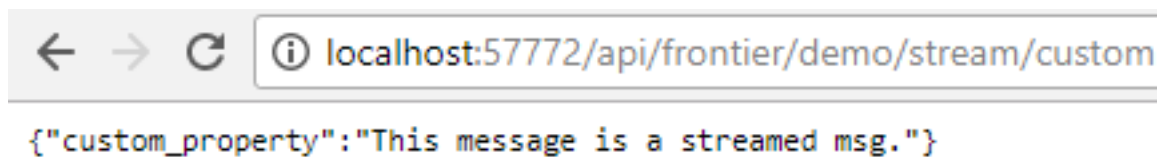
So let's see how it works, create a new route just like this:

```
<Route Url="/stream/custom" Method="GET" Call="GetCustomizedTextObject"></Route>
```

And another method to handle it.

```
ClassMethod GetCustomizedTextObject() As %DynamicObject  
{  
    set stream = ##class(%Stream.GlobalCharacter).%New()  
    do stream.Write("This message is a streamed msg.")  
  
    return {  
        "customproperty" : (stream)  
    }  
}
```

So that we get this:



This concludes the basic usage of Frontier. Until now we've seen only how to handle GET requests, next time we are going to cover

POST, more precisely, how to handle payloads.

Link for [Part 2: Handling payloads](#).

Link for [Part 3: Using the SQL API](#).

Link for [Part 4: Sharing data across router methods](#).

Thank for your patience!

[#Object Data Model](#) [#REST API](#) [#Tutorial](#) [#Caché](#) [#InterSystems IRIS](#)
[Check the related application on InterSystems Open Exchange](#)

Source

URL: <https://community.intersystems.com/post/frontier-abstraction-layer-rapid-rest-development-part-1-core-concepts>