

Article

[Chris Stewart](#) · Apr 25, 2017 5m read

Let's write an Angular 1.x app with a Caché REST backend - Part 9

In our [last lesson](#), we implemented a linkage to our WidgetAccessory class, and debugged some errors we encountered along the way. We now have our data being returned by REST, but what if we want to update or add new data to our application?

So far we have only used HTTP GET commands with our REST Services, we now have to implement PUT (which roughly corresponds to an Update) and a POST (which roughly corresponds to a Create. Author's note: there is a lot of writing online about why this statement isn't entirely correct, I'm not going to go into the detail here). However, before we start, we notice that we should probably do some refactoring, since we want to be interacting with Widgets. REST typically involves accessing a logical, human readable URL pattern, so we should really tie our Widget methods to a Widget URL. We could just add these into the REST.Dispatch class, but this will lead to a very bulky class. Luckily, we can set a forward from the Dispatch class to another class which also inherits from %CSP.REST. So with this in mind, we want to create a REST.Widget class, and route all widgetmaster/rest/widget/ calls to it.

```
⊞XData UrlMap [ XMLNamespace = "http://www.widgetsdirect.com/urlmap" ]
{
  ⊞<Routes>
    <Map Prefix = "/widget" Forward="REST.Widget" />
    <Route Url="/:name" Method="GET" Call="HelloWorld" Cors="false" />
  </Routes>
}
```

This will unpack the URL, and forward all calls onto the specified class (and will remove the prefix from the URL, so /widget/1 will just be forwarded as /1)

We are getting a little ahead of ourselves though. Before we start creating new Services, we should make sure we have a way to read a JSON string, and use it to update our Widget object. Since we have a toJSON() to convert an object instance to a JSON string, we will implement a fromJSON to do the reverse and update the properties of an object instance and %Save() it. We could do this in a very basic way, by unpacking all of the JSON properties and assigning them directly to the object, but this isn't very robust as it assumes that all properties will always be populated, and is also quite unmanageable for larger numbers of properties

```
⊞Method fromJSON(json as %String) As %String
{
    set jsonObj = json.%FromJSON()

    set ..Name           = jsonObj.Name
    set ..Description    = jsonObj.Description
    set ..Price          = jsonObj.Price
    set ..Quantity       = jsonObj.Quantity

    quit ..%Save()
}
```

So, instead, let's iterate over the properties of our JSON input, using the %GetIterator and %GetNext in a loop to unpack all of our properties, filter them (for example, we wouldn't want to try to update an Id value), and then assign these to the object instance using \$PROPERTY

```

Method fromJSON(json As %String) As %String
{
|
  set jsonObj = {}.%FromJSON(json)

  set propsIterator = jsonObj.%GetIterator()
  While (propsIterator.%GetNext(.key,.value)) {
    if ((key="Name") || (key="Description") || (key="Price") || (key="Quantity")) {
      Set $PROPERTY($this, key) = value
    }
  }

  quit ..%Save()
}

```

Much neater. We can now use this method to apply a JSON update to our object instance. Let's get back to our new REST.Widget class. Create the class, inheriting from the %CSP.REST superclass. We will then hook up the most common usage patterns to the appropriate verb

```

XData UrlMap [ XMLNamespace = "http://www.widgetsdirect.com/urlmap" ]
{
  <Routes>
  <Route Url="/" Method="GET" Call="GetAllWidgets" Cors="false" />
  <Route Url="/" Method="POST" Call="AddNewWidget" Cors="false" />
  <Route Url="/:widgetid" Method="GET" Call="GetWidgetById" Cors="false" />
  <Route Url="/:widgetid" Method="PUT" Call="UpdateWidgetById" Cors="false" />
  <Route Url="/:widgetid" Method="DELETE" Call="DeleteWidgetById" Cors="false" />
  </Routes>
}
}

```

With these 5 Services, we can implement a full CRUD system for our class. By GETing the default path, we can return all instances (for browsing/searching), or we can GET a specific record by its ID (useful for returning more detail on larger objects). We can PUT an update to a specified object instance, and we can POST a new object entirely to the default path. Finally, we can implement a DELETE call, which will perform some form of Deletion or Archiving to the specified object.

We can implement the GetAllWidgets first, as we already have the code in our REST.Dispatch class. We create the GetAllWidgets classmethod (with no params) and paste over the code from REST.Dispatch, removing it from that class when we're done.

```
ClassMethod GetAllWidgets() As %Status
{
    Set %response.ContentType="application/json"

    SET retObj = {}
    SET widgetAry = []
    &SQL(DECLARE WidgetCurs CURSOR FOR
        SELECT
            %Id
            INTO :Id
            FROM SQLUser.Widget
    )

    &SQL(OPEN WidgetCurs)

    For { &SQL(FETCH WidgetCurs)
        Quit:SQLCODE
        set widgetObj = ##class(User.Widget).%OpenId(Id)
        do widgetAry.%Push(widgetObj.toJSON(1))
    }
    &SQL(CLOSE WidgetCurs)
    SET retObj.Widgets = widgetAry

    WRITE retObj.%ToJSON()
    QUIT $$$OK
}
```

Next, we can implement the GetWidgetById. This is a very easy pattern to implement. We unpack the Id value passed in, use it to open the relevant object instance, and return the toJSON() output via a write. If the object fails to load, then we pass that error back out as a return value

```

ClassMethod GetWidgetById(WidgetId as %Integer) As %Status
{
    Set %response.ContentType="application/json"

    set tSC = $$$OK
    SET retObj = {}
    SET widgetAry = []

    // Safely retrieve object
    Kill %objlasterror
        set widgetObj = ##class(User.Widget).%OpenId(WidgetId)
        If '$IsObject(widgetObj) {
            // Object with this ID does not exist
            If $Data(%objlasterror) { Set tSC=%objlasterror }
        }
        If $$$ISERR(tSC) { Quit }
    // Set output to JSON representation of object
    SET retObj.Widget = widgetObj.toJSON(1)

    WRITE retObj.%ToJSON()
    QUIT tSC
}

```

Now, we move to using our new fromJSON method to update or add objects. These 2 classmethods are basically identical, with the UpdateWidgetById using the supplied ID value to open a specific object instance, while the AddNewWidget just performs a %New() to get a new object instance to write to. Once an object instance has been loaded, the fromJSON is run. This takes the contents of the Request as the JSON input - `Set updateJSON = %request.Content.Read()`. Any Save errors are passed back to the client. As a final step, the current state of the object is returned using the toJSON(). This is very useful when we start implementing these calls in our application, as we can instantly rebind the current state of the object to the display, without having to make a secondary call to a GET.

```

ClassMethod AddNewWidget() As %Status
{
    Set %response.ContentType="application/json"

    SET retObj = {}

    Kill %objlasterror
    set widgetObj = ##class(User.Widget).%New()
    If '$IsObject(widgetObj) {
        // Object with this ID does not exist
        If $Data(%objlasterror) { Set tSC=%objlasterror }
    }
    Set updateJSON = %request.Content.Read()

    Set tSC = widgetObj.fromJSON(updateJSON)

    If $$$ISERR(tSC) { Quit tSC }

    WRITE widgetObj.%ToJSON()
    QUIT $$$OK
}

```

```
ClassMethod UpdateWidgetById(widgetid as %Integer) As %Status
{
    Set %response.ContentType="application/json"

    SET retObj = {}

    Kill %objlasterror
    set widgetObj = ##class(User.Widget).%OpenId(widgetid)
    If '$IsObject(widgetObj) {
        // Object with this ID does not exist
        If $Data(%objlasterror) { Set tSC=%objlasterror }
    }
    Set updateJSON = %request.Content.Read()

    Set tSC = widgetObj.fromJSON(updateJSON)

    If $$$ISERR(tSC) { Quit tSC }

    WRITE widgetObj.toJSON().%ToJSON()
    QUIT $$$OK
}
```

We can test these operations out in our REST debugger. When the PUT or POST verb is selected, you should get an option to include a request Payload. We will include a full object for the POST (Create) and will update one field for our POST. On each call, we should get the current state of the object (including our changes) as a response. Note that in the first example (POST), the Id value is discarded, and instead comes from the %Save(), as we would expect

BODY

```
1 { "Id":"107",  
2  "Name":"Wacky Widget",  
3  "Description":"This widget is shockproof",  
4  "Price":"3.99",  
5  "Quantity":"85"}
```

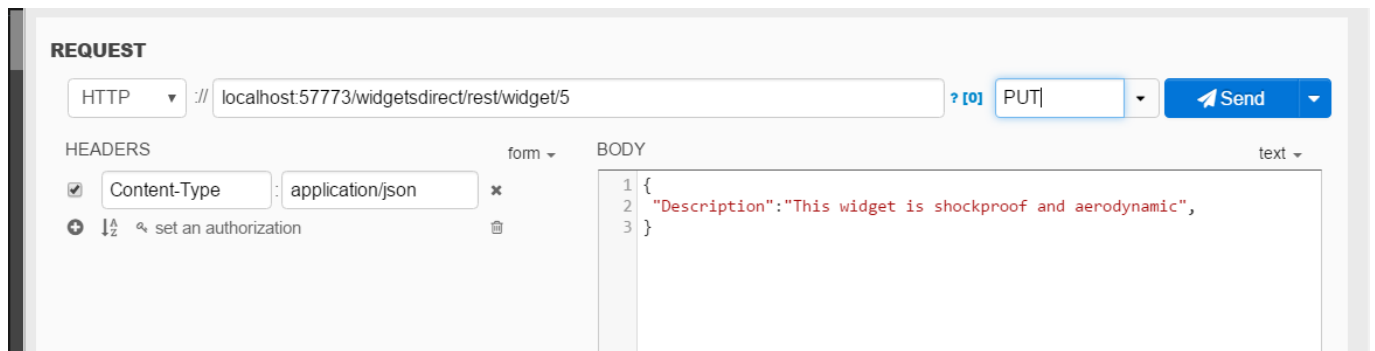
Text | **JSON** | XML | HTML Enable body's evaluation

BODY

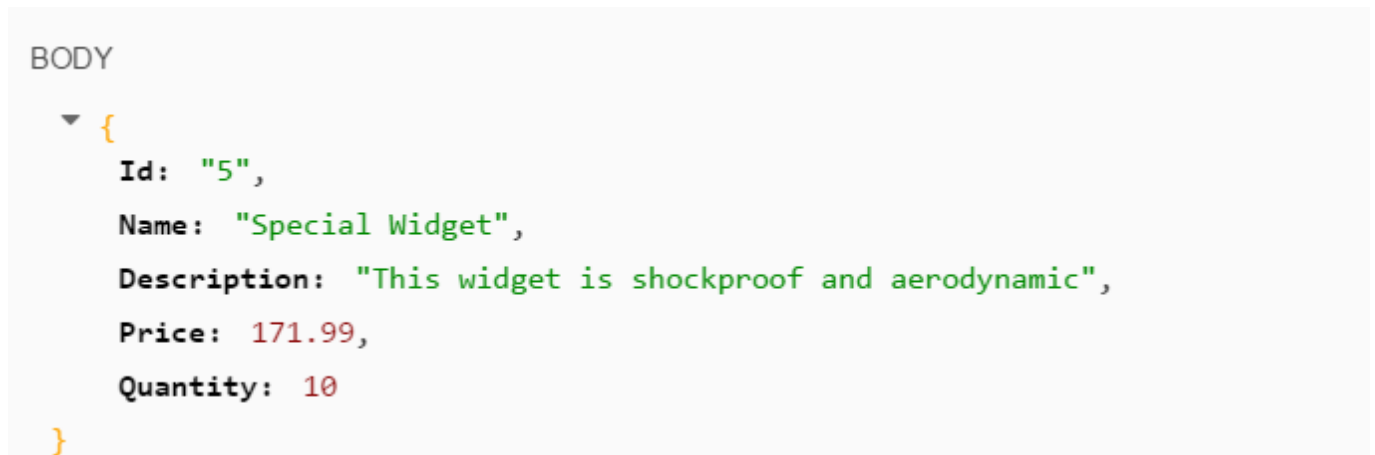
```
▼ {  
  Id: "6",  
  Name: "Wacky Widget",  
  Description: "This widget is shockproof",  
  Price: 3.99,  
  Quantity: 85  
}
```

[lines nums](#)

Now for our Description update



And we can see from the response that the Description has been updated, while all other fields remain the same



Let's check that our new and updated data is available on our Welcome page. First, however, we need to point the pageController at our new service to get the full list of Widgets (since they are no longer returned from our HelloWorld service). We implement a second \$http.get and assign the response to the Widgets array. If we fail in our call to the Widgets service, we set the Widgets array to empty, to prevent warnings from anything trying to read the array.

```
angular
```

```
.module('WidgetsDirect', ['ngMaterial'])
.controller('PageController', ['$q', '$scope', '$timeout',

$scope.message = "";
$scope.sortAsc = true;

$http.get('/widgetsdirect/rest/Chris').then(
  function(response) { //success
    $scope.message = response.data.Message;
  }
, function(response) { //failure
  $scope.message = "Couldn't get data :(";
  }
);


$http.get('/widgetsdirect/rest/widget/').then(
  function(response) { //success

    $scope.widgets = response.data.Widgets;
  }
, function(response) { //failure
  $scope.widgets = [];
  }
);
```


A quick reload and we can see our updated data

Welcome to Widgets Direct Chris


Filter REORDER




Widgets Direct - 6
Wacky Widget - \$3.99
There are 0 compatible accessories
This widget is shockproof




Widgets Direct - 5
Special Widget - \$171.99
There are 1 compatible accessories
This widget is shockproof and aerodynamic




Widgets Direct - 4
Music Widget - \$170.99
There are 1 compatible accessories
This widget can autotune all brass and string instruments



Widgets Direct - 3
Racing Widget - \$17.99
There are 0 compatible accessories
This widget records average speed and lap time



Widgets Direct - 2
Woodland Widget - \$7.99
There are 2 compatible accessories
This widget identifies plant and tree species



Widgets Direct - 1
Waterproof Widget - \$12.99
There are 3 compatible accessories
This widget is waterproof to 100m depth for a time of up to 7 hours

Today we:

1. Implemented a new Widget REST service, with CRU operations (we don't want to do a Delete just yet)
2. Implemented a forwarder from the REST.Dispatch to our new REST.Widget class
3. Created a fromJSON() to assign the properties of a JSON string to our Widget object
4. Implemented a GetAllWidgets REST Service
5. Implemented a GetWidgetById REST Service
6. Implemented an AddNewWidget REST Service
7. Implemented an UpdateWidgetById REST Service
8. Refactored our controller to read from the new Service

Next time we will:

- Implement a fromJSON() for WidgetAccessory
- Create a basic form to add new Widgets

This article is part of a multi-part series on using Angular on top of Caché REST services. The listing of the full series can be found at the [Start Here](#) page

[#Angular](#) [#CSP](#) [#JavaScript](#) [#REST API](#) [#Frontend](#) [#Caché](#)

Source

URL:<https://community.intersystems.com/post/lets-write-angular-1x-app-cach%C3%A9-rest-backend-part-9>