INTEGRATING A REST-BASED SERVICE WITH K2

Based on the REST broker in K2 Appit 1.4

6/17/2016

K2 allows you to interact with REST endpoints by using the REST SmartObject Broker to expose a particular REST endpoint as a service instance, create SmartObjects that utilize the service objects exposed by the service instance, and then optionally build SmartWizards that expose these SmartObjects as wizards so that workflow designers can easily drag and drop methods from the REST-based SmartObjects into their workflows.

This article demonstrates this integration with an end-to-end scenario that involves exposing REST Endpoints provided by the Swagger PetStore example as wizards that can be used in K2 workflow design tools.

Notes

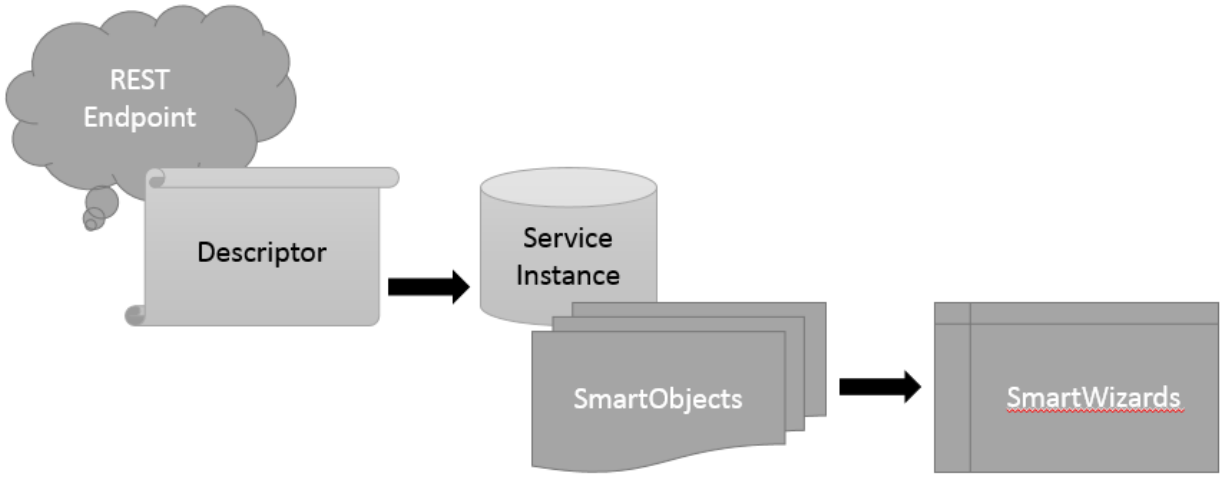
1. This article assumes that you are already familiar with SmartObjects, Service Instances and building workflows with K2 Studio as well as REST services and basic JSON.
2. K2 Appit does not include the ability to generate and register custom SmartWizards. That portion of this article only applies to those running K2 blackpearl.

At a high level the REST integration comprises of the following components:

- **Descriptor File**: The file that the REST Broker Service Instance uses to describe entities and methods for the REST endpoint. This file is a JSON (JavaScript Object Notation) file in Swagger format that is constructed based on the REST endpoint entities and methods. For this scenario we will create a Descriptor file that exposes LinkedIn REST endpoints.
- **OAuth Resource**: The OAuth resource used in the Service Instance Authentication Mode settings to connect to the REST endpoint. The OAuth resource is only necessary if the endpoint requires OAuth authentication; the PetStore example does not require OAuth.
- **Service Instance**: The Service Instance of the REST Endpoint Broker which uses the descriptor file to connect to a particular REST endpoint (required) and uses the OAuth resource for authentication (optional).
- **SmartObjects**: SmartObjects that are generated based on the Service Objects exposed by the Service Instance. These SmartObjects can then be used in SmartForms, workflows or any other component that is able to interact with SmartObjects.
- **SmartWizards:** SmartWizards expose SmartObjects as workflow designer wizards, which allows workflow designers to work with the endpoint declaratively. You do not need to create SmartWizards if you do not wish to expose the REST SmartObjects as wizards to workflow designers. The SmartObjects can still be used as-is on the workflow designer using the SmartObject event wizard, but creating SmartWizards will make it easier for workflow designers to interact with the endpoints. SmartWizards can be used in the browser-based workflow design tool as well as the thick-client design tools like K2 Studio and K2 for Visual Studio.

The diagram below illustrates the relationship between these components:

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.



THE DESCRIPTOR FILE

A text-based descriptor file describes the REST service to K2. This descriptor file is formatted using the Swagger (http://swagger.io/) JavaScript Object Notation (JSON) spec, based on an Entity and Path structure. A descriptor is required for each REST service, specifically the endpoint's base URI, in order to register that endpoint for K2 integration. Each descriptor is associated with a separate Service Instance, and the Service Instance uses the descriptor file to understand the entities and methods exposed by the REST service.

Note

The distinction between base URI and endpoint (aka path) is important. The base URI must be the same for all endpoints in the descriptor. The descriptor file also represents the service instance boundary. If you have two REST services but they differ in their base URI, for example https://api.example.com and https://docs.example.com, they must be two separate descriptor files and, because of that, two service instances.

At the most basic level a descriptor is structured as follows:

```
{
    "swagger": "2.0",
    "host": "petstore.swagger.io",
    "basePath": "/v2",
    "schemes": [ "http" ],
    "paths": {
            …
    },
    "definitions": {
            …
    }
}
```

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

The descriptor file can contain only one Base Path but can have multiple definitions. Definitions, sometimes referred to as entities, can be referenced in other definitions. You can have as many endpoints off of the same base path as you like, with each method being a separate path. The supported REST methods are:

- GET
- POST
- PUT
- PATCH
- DELETE

Note

You must fully describe all entities and properties that are part of the REST endpoint. You can have more than what's used in the particular REST endpoint but you cannot have less. If you have not described a property or entity returned in the REST payload, the broker will report an error.

DESCRIPTOR EXAMPLE

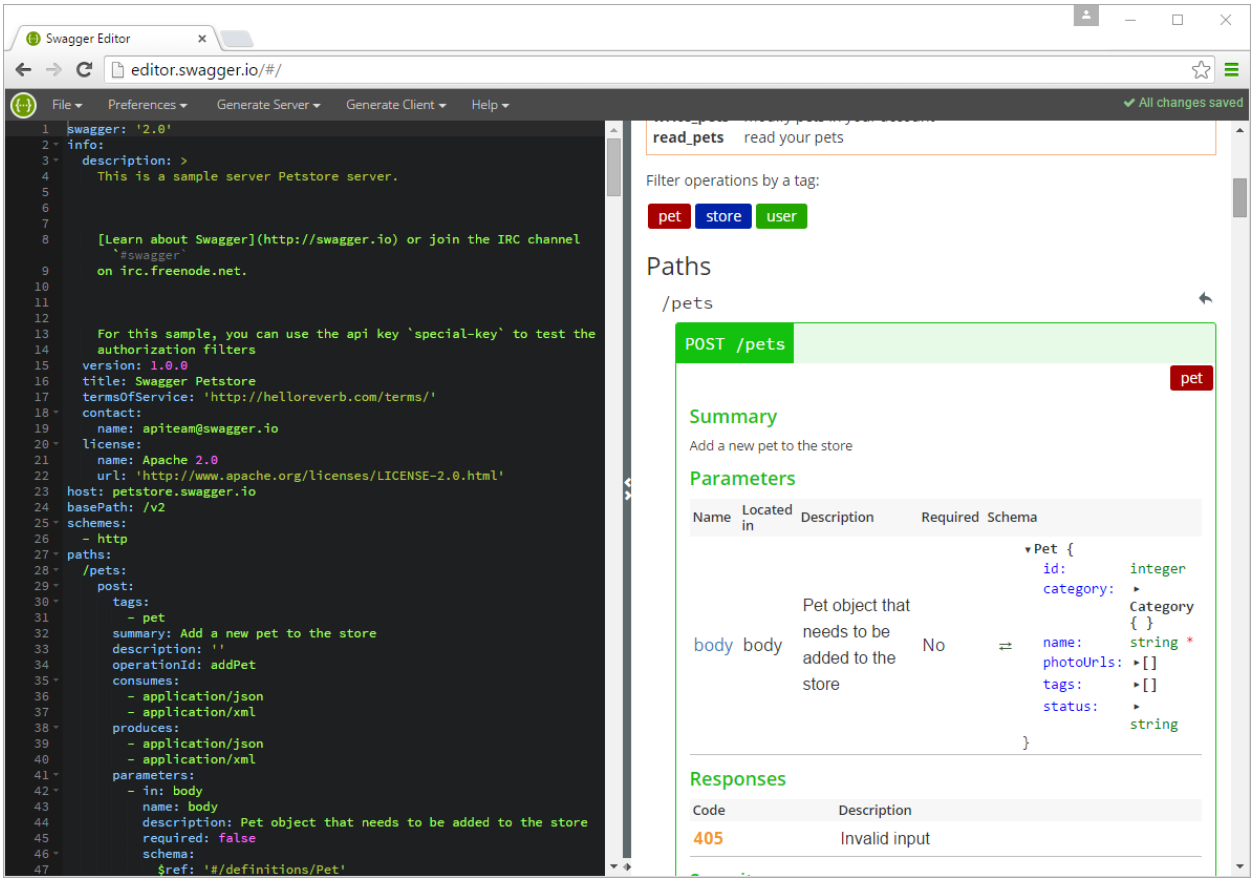
For this scenario we will be using the PetStore REST API.

Note

Swagger makes occasional changes to the PetStore. Your results using the resources provided in this article may vary, but if you have a specific PetStore scenario in mind, read the documentation carefully in order to determine whether your scenario is possible.

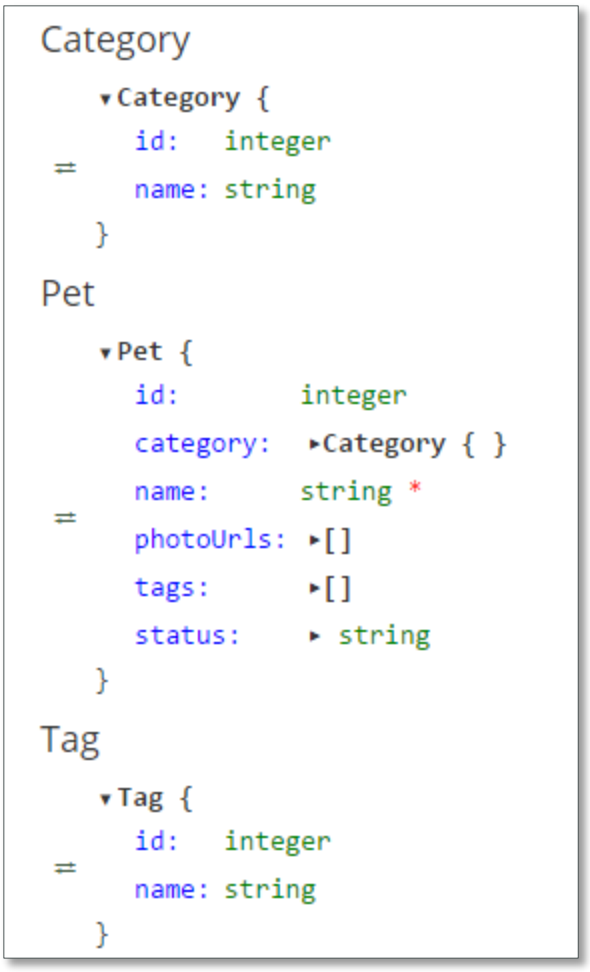
When using the editor.swagger.io/#/ editor, you see a visual representation of your descriptor file. The first path in the descriptor file is a POST for adding a pet to the store:

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.



As you scroll down the right pane you see the rest of the paths and then the models (aka entities) appear.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.



Each entity has a name and a properties collection containing the properties of the entity and each of their data types. Here's an example of the Pet entity which includes two required items, **name** and **photoUrls**, and referenced properties, **Category** and **tags**.

```
            "Pet" : {
                  "type" : "object",
                  "required" : ["name", "photoUrls"],
                  "properties" : {
                        "id" : {
                              "type" : "integer",
                              "format" : "int64"
                        },
                        "category" : {
                              "$ref" : "#/definitions/Category"
                        },
                        "name" : {
                              "type" : "string",
                              "example" : "doggie"
                        },
                        "photoUrls" : {
```

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

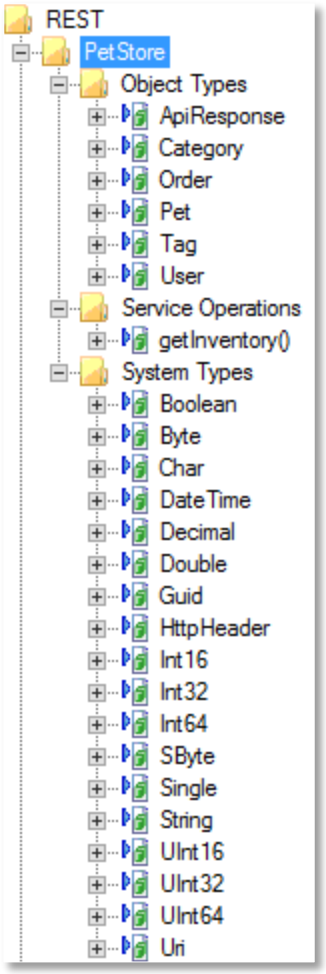
```
                              "type" : "array",
                              "xml" : {
                                    "name" : "photoUrl",
                                    "wrapped" : true
                              },
                              "items" : {
                                    "type" : "string"
                              }
                        },
                        "tags" : {
                              "type" : "array",
                              "xml" : {
                                    "name" : "tag",
                                    "wrapped" : true
                              },
                              "items" : {
                                    "$ref" : "#/definitions/Tag"
                              }
                        },
                        "status" : {
                              "type" : "string",
                              "description" : "pet status in the
store",
                              "enum" : ["available", "pending", "sold"]
                        }
                  },
                  "xml" : {
                        "name" : "Pet"
                  }
            },
```

You can see that it has a string property called **status** which is an enumeration, which can be one of three values – available, pending or sold. Referenced entities, such as Category and tags, must be defined before they can be used. References to other entities are described as `"$ref": "#/definitions/Category"`

Note Remember to define referenced entities first before trying to reference them in another entity. This is why the **Category** and **Tag** entities appear before the **Pet** entity.

When you generate SmartObjects for these entities, each entity becomes its own SmartObject with properties, methods and parameters. System types are objects that are provided for every instance and can be used when working with the entities.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.



As for the endpoints, you must have at least one path in your descriptor file in order for the service instance to do anything with the SmartObjects. You do not need to define any entities, however, and all methods in this case that are not tied to an entity are placed in the Service Operations folder.

When you reference an entity in the response, that method gets attached to that entity by the REST broker. For example, the first path in the PetStore descriptor file is the **addPet** POST. This allows you to add a pet to the store. This method takes one parameter, **body**, which is a complex type that conforms to the **Pet** entity.

```
        "/pet" : {
                "post" : {
                     "tags" : ["pet"],
                     "summary" : "Add a new pet to the store",
                     "description" : "",
                     "operationId" : "addPet",
                     "consumes" : ["application/json",
"application/xml"],
                     "produces" : ["application/xml",
"application/json"],
                     "parameters" : [{
```

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

```
                                    "in" : "body",
                                    "name" : "body",
                                    "description" : "Pet object that
needs to be added to the store",
                                    "required" : true,
                                    "schema" : {
                                          "$ref" : "#/definitions/Pet"
                                    }
                              }
                        ],
                        "responses" : {
                              "405" : {
                                    "description" : "Invalid input"
                              }
                        },
                        "security" : [{
                                    "petstore_auth" : ["write:pets",
"read:pets"]
                              }
                        ]
                  },
```

REGISTERING THE SERVICE INSTANCE

There are multiple steps in registering a service instance with a descriptor file.

1. If the service requires authorization, create the necessary integration portion on the target site, resulting in a Client ID and a Client Secret for OAuth. See the K2 KB article How to configure OAuth in K2 (http://help.k2.com/KB001702) for more information on creating OAuth resources. The PetStore example does not require authorization.
2. Configure your service instance in order to successfully call and write to the REST service. This step most likely involves the use of 3rd party monitoring tools such as Fiddler or WireShark. This example uses Fiddler. For Appit customers, there is no direct method of capturing traffic from the K2 server.
3. Generate SmartObjects and test the service.

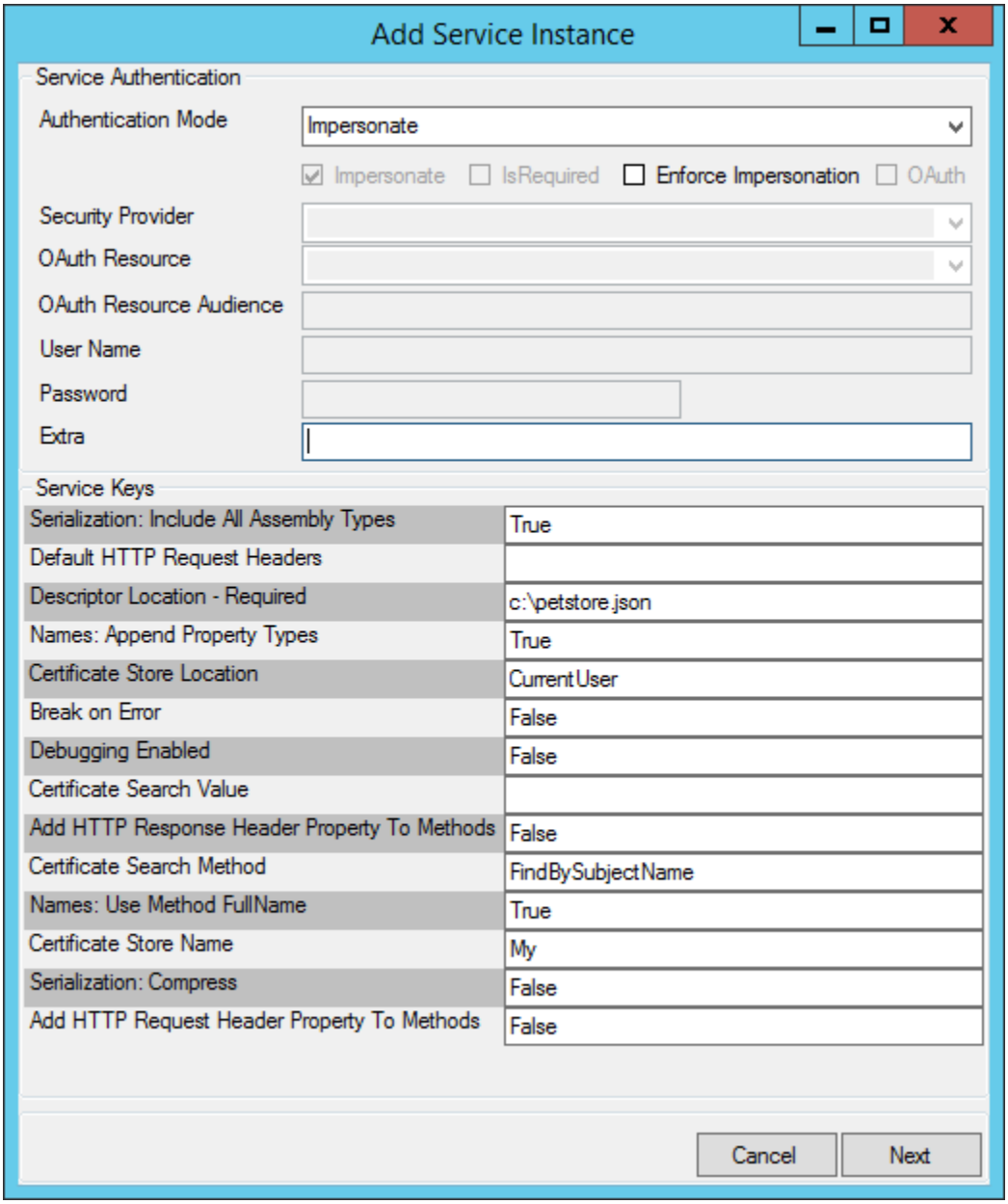
Since the PetStore example does not require OAuth authorization to connect to the Swagger service, you can leave the Authentication Mode as Impersonate (the default setting). Then type in the path to the PetStore.json file stored locally on the server and click **Next**.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

Note

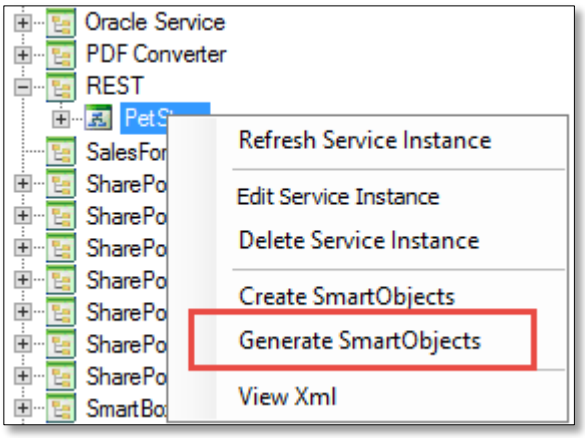
For Appit customers, you must specify a **Descriptor Location** that is accessible on the internet. If you're using a service that requires authentication, your descriptor location can be of the same authentication mechanism or it can be anonymous. However, it cannot be a different authentication mechanism (other than anonymous) than the service requires.

For the PetStore example you can use the following URL for the **Descriptor Location**: http://k2.com/help/resources/samples/petstore.json



© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

Once the instance is created you need to generate SmartObjects. Right-click the instance and select **Generate SmartObjects**. For Appit customers, use the K2 Management website and check the Create SmartObjects option at the bottom of the dialog when editing the service instance. This calls the same API as the Generate SmartObjects action does.



When using the SmartObject Service Tester tool to do this, your SmartObjects are created in the REST category unless you create a new service type category that is based on the REST broker but called something unique (see Appendix B: Creating a Custom Service Category). Browse to the correct category under the SmartObjects node and execute a method on a SmartObject. For Appit customers, you do not have the ability to create a new service category so all REST-based SmartObjects reside in the REST category.

Warning

OAuth Tip: If your service instance uses OAuth, and you are testing your descriptor by using the SmartObject Services Tester tool, you are initially redirected to login to the OAuth resource. From that point forward you may not be required to authenticate again.

However, when a workflow executes a SmartObject method, its identity is as the K2 service account and you must re-authorize using that URL before the workflow can run without error. This URL can be found in the error message in K2 Workspace:

Source	Description
Search Company	OAuth token is expired and requires reauthorization. User: K2 Server Account URL: https://www.linkedin.com/uas/oauth2/authorization?response_type=code&client_id=75ygagcn7renlz&redirect_uri=https://k2.denallix.com/identity/token/oauth/2&state=PrimaryCredentialID%3d94669253-158b-4614-b23f-864baacf9cef%7cResourceID%3d72677226-61bc-461c-82d3-fccfb6b77383

Logging in with this URL automatically invalidates any other OAuth token issued from your login, meaning the identity used with the SmartObject Services Tester tool will now prompt you to authenticate with the line-of-business system again (in turn, invalidating the K2 Service account's token).

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

TESTING THE API USING FIDDLER

The best way to determine if your REST service instance is configured correctly is to examine the payloads from the K2 server to the endpoint using a tool like Fiddler. To do this, you must setup a proxy for traffic from the K2 server to be captured by Fiddler.

To do this following these steps:

1. Stop K2 blackpearl Server
2. Open "C:\Program Files (x86)\K2 blackpearl\Host Server\Bin\K2HostServer.exe.Config"
3. Add the following to the <system.net> section.

> **Note** These sample values are from the K2 Denallix Core's Fiddler configuration – you must change these values to reflect the values in your environment.

```
<defaultProxy enabled="true" useDefaultCredentials = "true">

    <proxy autoDetect="false" bypassonlocal="false"
proxyaddress="http://127.0.0.1:9999" usesystemdefault="false" />

</defaultProxy>
```

4. Restart K2 blackpearl Server

When you are finished examining the traffic from the K2 server, set the enabled flag to false. If you do not do this and Fiddler is not running, the K2 server may not be able to accept requests.

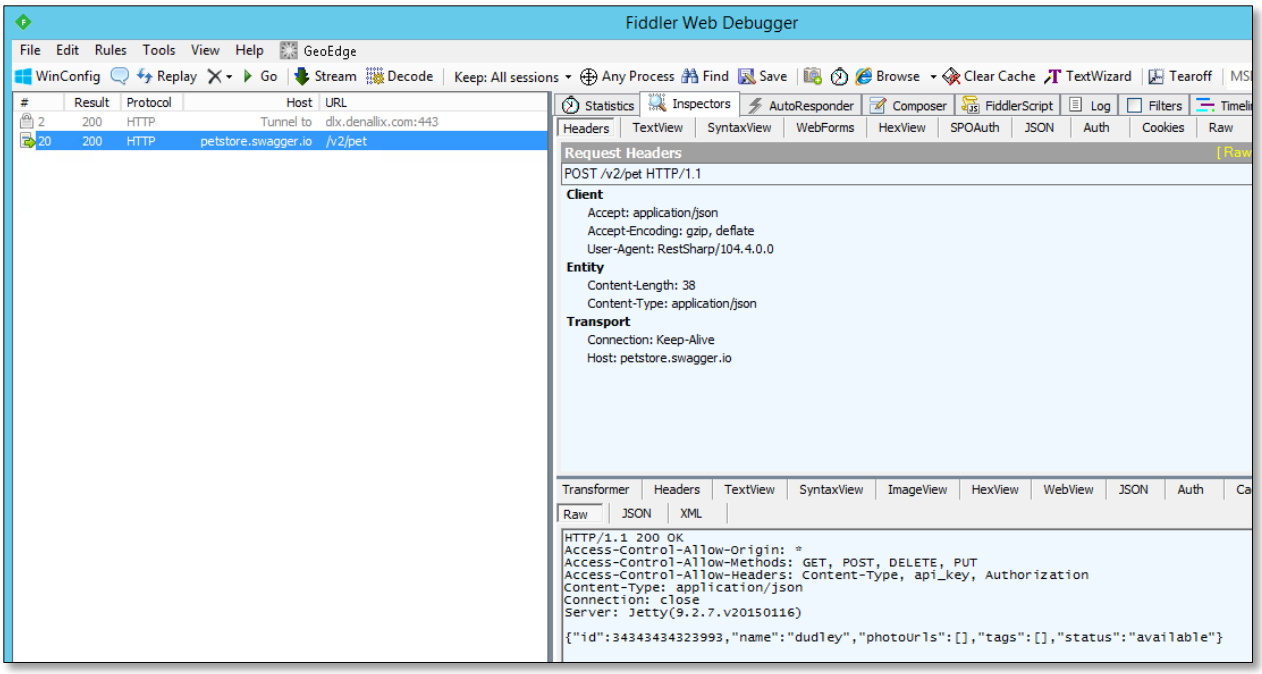
```
<defaultProxy enabled="false" useDefaultCredentials = "true">

    <proxy autoDetect="false" bypassonlocal="false"
proxyaddress="http://127.0.0.1:9999" usesystemdefault="false" />

</defaultProxy>
```

Once you have confirmed that you are capturing K2 server traffic, execute a SmartObject method. You can see in the following Fiddler trace that the call was made to the PetStore API /v2/pet.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.



Note Most endpoints are encrypted using SSL. To decrypt this traffic in Fiddler go to **Tools** > **Fiddler Options**. On the HTTPS tab check the **Decrypt HTTPS traffic …from all processes**. For each session that's encrypted you'll still have to click the button to decrypt it but after that you can see the REST payload.

Use the following iterative method to finalize the foundation of your solution:

1. Make changes to your descriptor file
2. Refresh your service instance
3. Generate your SmartObjects
4. Examine the web traffic using Fiddler

Once you're happy that your descriptor file has fully described everything you want to do with the REST endpoint, you're ready to start creating your SmartWizards (and if you don't need to call endpoint methods from workflows, you're done! For Appit customers, since there is currently no method to add custom SmartWizards to your environment, you can read ahead but cannot implement these at the moment).

CREATING SMARTWIZARDS

Use the Help topic Custom K2 SmartWizards to familiarize yourself with how to create SmartWizards.

Note: You must have K2 blackpearl 4.7 or 4.6.11 with a coldfix in order to successfully run the example SmartWizards.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

For the purposes of this article there are a few key points in developing your SmartWizards.

1. Decide how many SmartWizards you're going to create. You'll typically have one for each method that you need to expose to workflow designers. Not every SmartObject method requires a SmartWizard: you may have more methods on the service that can be called directly as SmartObject methods which do not have a corresponding SmartWizard.
2. Create icons for each wizard in the following sizes: 16x16 icon file (.ico), 32x32 image file (.png) and 64x64 image file (.png). Sizes are in pixels. If you're not going to use K2 Studio wizards, you can skip the 16x16 .ico file.
3. Determine if you'll have multiple instances of your REST service, and if the workflow designers must select the appropriate instance depending on the scenario. There may be scenarios where your REST endpoint connects to different systems or as different identities depending on the data that is stored in the system, which both would require more than a single instance. When designing SmartWizards, you either hardcode the service instance name or allow the workflow designer to select the instance, which will affect how you design your SmartWizards. In this example we'll include controls to select the instance. Note: You may want to create a separate category for your collection of service instances. For more information about how to do this see Appendix B: Creating a Custom Service Category.

When you have decided what methods you want to expose as SmartWizards it's time to create the folders for those categories and start working on the XML files for those.

It's easiest to develop the wizards for K2 Studio (and K2 for Visual Studio) first, and then deploy them as wizards that can be used in K2 Designer and K2 for SharePoint. The SmartWizards folder is located at [install drive]:\Program Files (x86)\K2 blackpearl\Bin\DesignTemplates\SmartWizards

In this folder you will see many SmartWizards. In fact, all of the wizards for SharePoint 2013 are SmartWizards, and you can study the layout.xml files there in order to learn by example how to do certain actions that you see in those wizards. The first level subfolder in the SmartWizards folder displays as a category in K2 Studio while the second level folder is the actual wizard folder containing the layout.xml file, the wizard.ico file and any additional resources like .png files that might be used in your wizard pages.

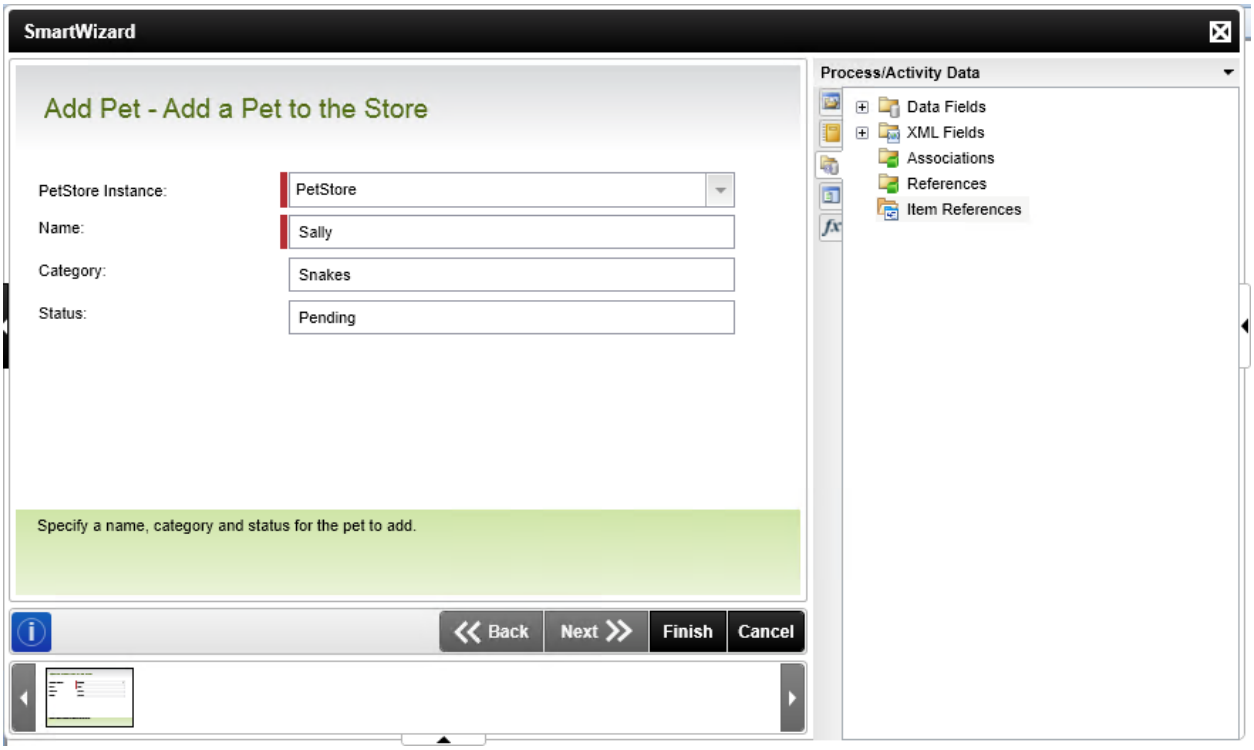
In this example we will create the main LinkedIn folder containing the following folders:

- Add Pet
- Get Pet Reference
- Update Pet
- Delete Pet

These four wizards are the wizards that workflow designers will be able to use in their workflows.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

For the first wizard, **Add Pet**, the final result looks like this:



The wizard is looking for a service type called REST and fills the dropdown with instances of that service type when the wizard loads (if there is only one it is auto-selected – this is part of the SmartWizard framework).

Let's take a closer look at how this happens.

This wizard is a single page wizard with the following page definition:

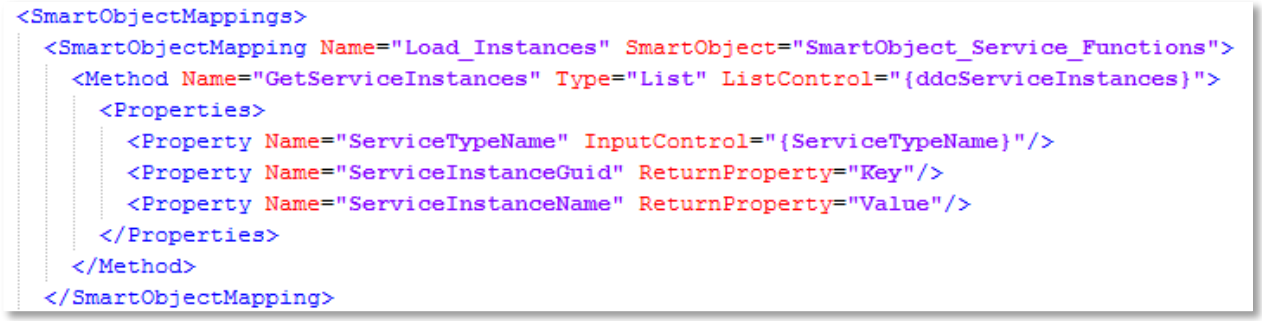
```
<SmartWizard>
  <WizardPages>
    <WizardPage Name="MainPage" Load="" Unload="" HelpID="">
      <Header Text="Add a Pet to the Store"/>
      <Body>
        <Label Name="lblServiceInstance" Text="PetStore Instance:" Top="7" Left="0" Width="150" Height="30" altTop="17" altLeft="0" altWidth="150" altHeight="30"/>
        <ComboBox Name="ddcServiceInstances" ResultType="ServiceInstance" Required="True" Click="LoadInstance" Load="LoadInstances" Top="0" Left="170" Width="320"
        Height="25" altTop="17" altLeft="180" altWidth="320" altHeight="25"/>
        <Label Name="lblName" Text="Name:" Top="33" Left="0" Width="150" Height="30" altTop="45" altLeft="0" altWidth="150" altHeight="24"/>
        <K2TextBox Name="txtName" ResultType="None" WatermarkText="Specify a Name" Required="True" Top="30" Left="170" Width="320" Height="24" altTop="45" altLeft="180"
         altWidth="320" altHeight="24" MaxLength="50"/>
        <Label Name="lblCategory" Text="Category:" Top="63" Left="0" Width="150" Height="30" altTop="73" altLeft="0" altWidth="150" altHeight="24"/>
        <K2TextBox Name="txtCategory" ResultType="None" WatermarkText="Specify a Category" Required="False" Top="60" Left="170" Width="320" Height="24" altTop="73"
        altLeft="180" altWidth="320" altHeight="24" MaxLength="50"/>
        <Label Name="lblStatus" Text="Status:" Top="93" Left="0" Width="150" Height="30" altTop="101" altLeft="0" altWidth="150" altHeight="24"/>
        <K2TextBox Name="txtStatus" ResultType="None" WatermarkText="Specify a Status (Available, Pending, Sold)" Required="False" Top="90" Left="170" Width="320"
        Height="24" altTop="101" altLeft="180" altWidth="320" altHeight="24" MaxLength="50"/>
        <K2Data Name="ServiceTypeName" Text="SourceCode.SmartObjects.Services.Endpoints.Rest"/>
        <K2Data Name="ServiceInstanceName" Text=""/>
        <K2Data Name="SerializedCategory" Text=""/>
        <K2Data Name="SerializedPet" Text=""/>
      </Body>
      <Footer Text="Specify a name, category and status for the pet to add."/>
    </WizardPage>
  </WizardPages>
```

The **ddcServiceInstances** control has a **Load** method of **LoadInstances**. This method is the first method defined under **Events**.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

```
<Runtime>
  <RuntimeSteps>
    <Step ID="1" Name="SerializeCategory" Event="SerializeCategory"/>
    <Step ID="2" Name="SerializePet" Event="SerializePet"/>
    <Step ID="3" Name="AddPet" Event="AddPet"/>
  </RuntimeSteps>
</Runtime>
<Events>
  <Event Name="LoadInstances" Type="SmartObjectMapping" Target="Load_Instances"/>
  <Event Name="LoadInstance" Type="SmartObjectMapping" Target="Load_Instance"/>
  <Event Name="SerializeCategory" Type="SmartObjectMapping" Target="Serialize_Category"/>
  <Event Name="SerializePet" Type="SmartObjectMapping" Target="Serialize_Pet"/>
  <Event Name="AddPet" Type="SmartObjectMapping" Target="Add_Pet"/>
  </Events>
```

The **Load_Instances** method is a **SmartObjectMapping** that runs a SmartWizard framework method to load instances, passing in the **ServiceTypeName** K2Data field which is also defined in the wizard page above.

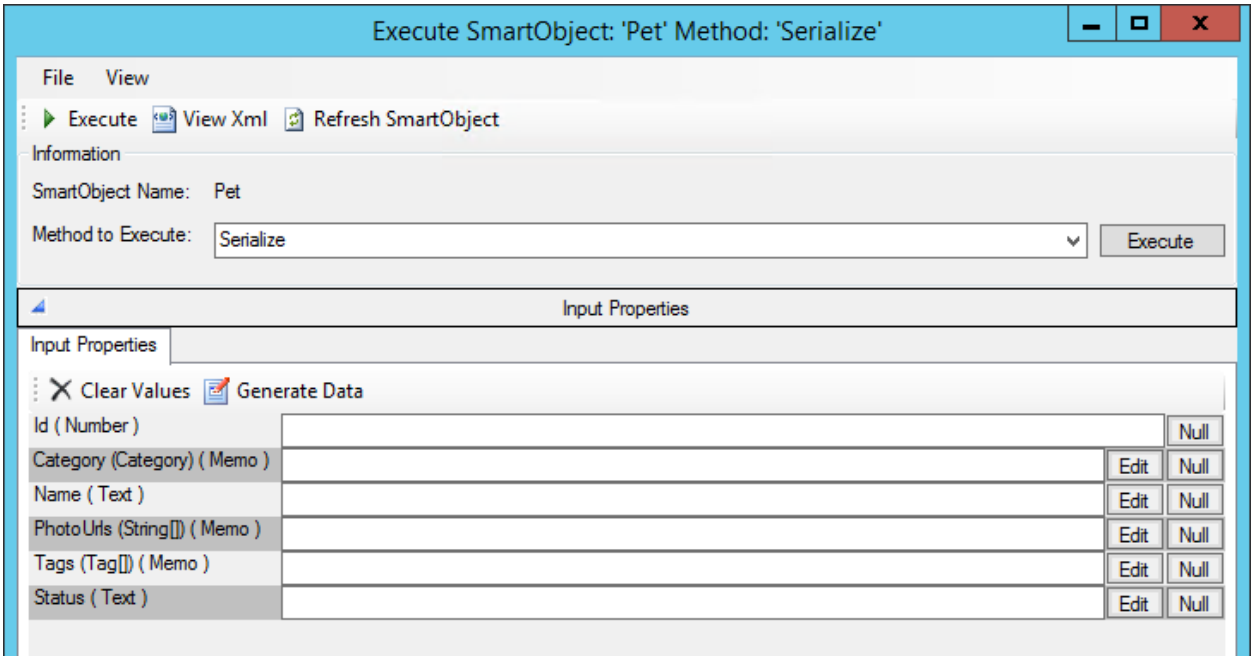


```
<SmartObjectMappings>
  <SmartObjectMapping Name="Load_Instances" SmartObject="SmartObject_Service_Functions">
    <Method Name="GetServiceInstances" Type="List" ListControl="{ddcServiceInstances}">
      <Properties>
        <Property Name="ServiceTypeName" InputControl="{ServiceTypeName}"/>
        <Property Name="ServiceInstanceGuid" ReturnProperty="Key"/>
        <Property Name="ServiceInstanceName" ReturnProperty="Value"/>
      </Properties>
    </Method>
  </SmartObjectMapping>
```

The **ServiceTypeName** is set to **SourceCode.SmartObjects.Services.Endpoints.Rest.RestBroker** since that is the name of the REST service type. If you created a separate category for your REST broker to only contain **PetStore** instances, you would call the ServiceTypeName **PetStore**. The best way to figure out what the service type names are is to click **View Xml** in the SmartObject Services Tester tool or look in the SmartBroker.ServiceType table in the K2 database.

In the Add Pet layout.xml file, there are two different serializations that must occur before the AddPet method can be called, namely to serialize the Category and to serialize the body containing the pet's information. This information in this example is **Name**, **Category** and **Status**. You could also extend this by capturing **PhotoUrls** and **Tags**. These are all optional items on the **Serialize** method of the **Pet** SmartObject.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.



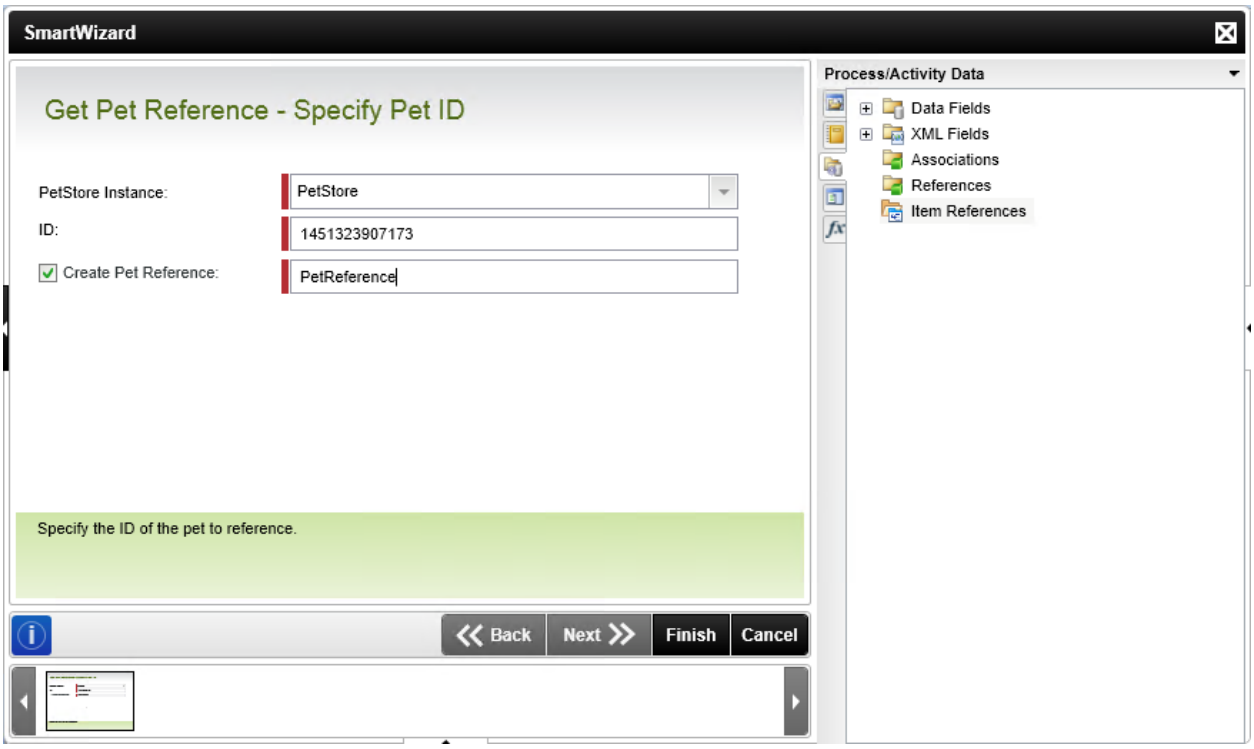
SAVING AN ITEM REFERENCE

With SmartWizards you can also create and use item references. This is powerful because you're giving workflow designers the ability to use entity properties from the REST service within their workflow design. For example, when creating an item or searching for something, it returns many properties about that item, and those are stored in the item reference. Actually just the ID is stored, but all properties can be retrieved using the ID.

Important Note on Functionality: For the **PetStore** example, there is not an easy way to create an item reference in the **Add Pet** wizard because the **AddPet** method doesn't return the ID of the newly-added pet. In order to make this sample work, you must manually search for the ID of the new pet and then use that ID in the **Get Pet Reference** wizard to return a reference to the pet created. This is a limitation of the way the PetStore service works. If you're designing your own REST web service and want to use it in your workflows, you must return the ID of the item or items created so it can work with the Item Reference framework in K2.

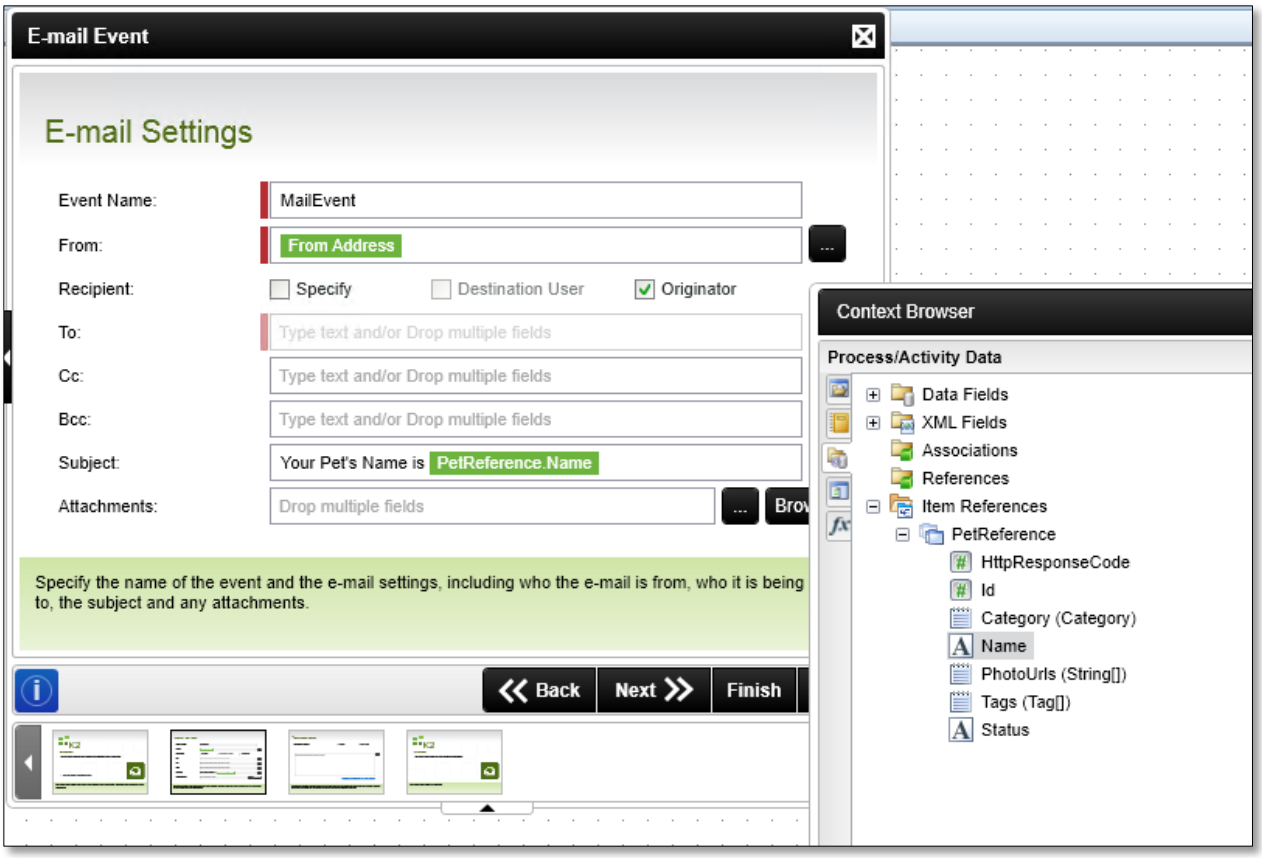
The **Get Pet Reference** takes in an ID and generates an item reference from it.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.



When configuring a step after the **Get Pet Reference** wizard, you can see the **Item References** node in the **Context Browser** is populated with the **PetReference**.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.



In the example above, the workflow designer can do something with the pet's properties in any event after the item reference is created. Note here that any property that has a type name in it in parentheses after the property name (such as **Category (Category)**) must be deserialized first before it can be used. For more information about serialization and deserialization see Using the Service Brokers topic in the K2 blackpearl User Guide.

To create an item reference in a SmartWizard you specify the reference in the References section at the end of the wizard layout.xml file:

Then you save that reference in the SmartObject mapping that performs the action at runtime, in this case retrieving the pet by ID:

```
<SmartObjectMapping Name="GetPet_ById" SmartObject="{ServiceInstanceName}_Pet">
  <Method Name="Read_GetPetById" Type="Read" SaveReference="PetReference">
    <Properties>
      <Property Name="petId_Read_GetPetById" InputControl="{txtID}"/>
    </Properties>
  </Method>
</SmartObjectMapping>
```

You must also add the **DefautLoad** property of the reference in the **References** section of the layout.xml file.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

```
<References>
  <Reference Name="PetReference" SystemName="{txtReference}">
    <DefaultLoad Name="Read_GetPetById" Type="Read">
      <Properties>
        <Property Name="petId_Read_GetPetById"/>
      </Properties>
    </DefaultLoad>
  </Reference>
</References>
```

To use the item reference in a subsequent wizard, you would also add the reference in the **References** section.

To add a dropdown list of pet references to a wizard, use the **K2ReferenceList** control and specify the name of the **ServiceType** and **ServiceObjectType**.

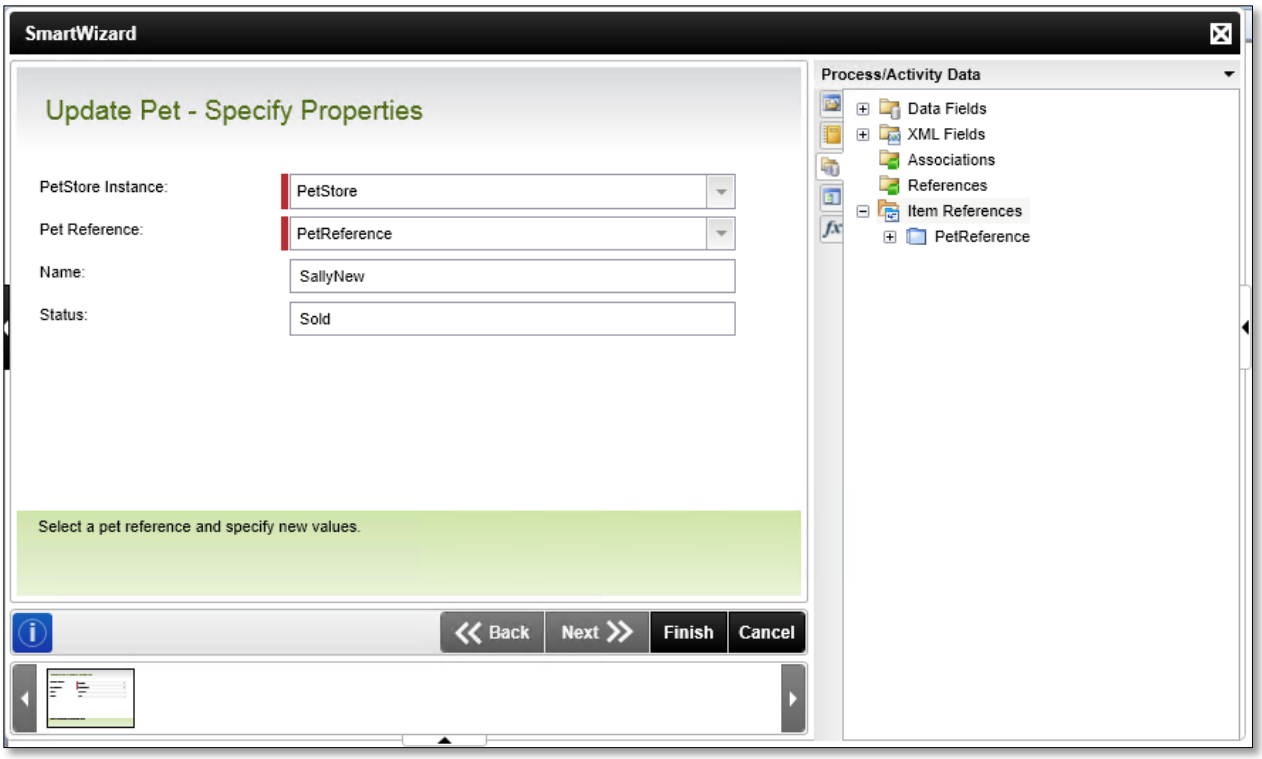
```
<K2ReferenceList Name="petreflist" Top="0" Left="160" Width="320" Height="24" altTop="73"
altLeft="170" altWidth="320" altHeight="24" Required="True" ServiceType=
"SourceCode.SmartObjects.Services.Endpoints.Rest.Broker" ServiceObjectType="Pet" />
```

To load the reference at some point during your runtime execution depending on your scenario. There may be some serialization methods that need to be called in sequence in order to serialize the response back to the REST endpoint. The key point here is that you specify the **LoadReference** and the **UseReference** attributes in order to load and use a previously-created reference.

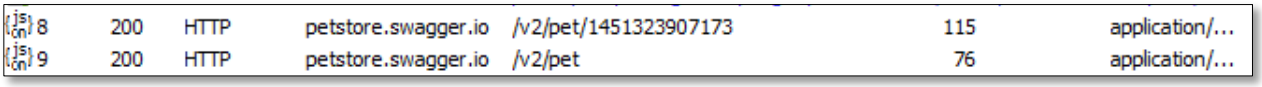
```
<SmartObjectMapping Name="Serialize_Pet" SmartObject="{petreflist}">
  <Method Name="Serialize" Type="Read" LoadReference="PetReference" UseReference="True">
    <Properties>
      <Property Name="Id" Load="True" MappingProperty="Id"/>
      <Property Name="Name" InputControl="txtName"/>
      <Property Name="Status" InputControl="txtStatus"/>
      <Property Name="Id" Load="True" MappingProperty="Id"/>
      <Property Name="Serialized_Item__String_" ReturnControl="{SerializedPet}" />
    </Properties>
  </Method>
</SmartObjectMapping>
```

In the PetStore example, the **Update Pet** wizard uses the previously created **PetReference** item reference to update the pet's **Name** and **Status**.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.



After running through this wizard and executing the workflow, the PetStore API is called once to get a reference to the existing pet, and then again to update the pet's name and status, as you can see in the Fiddler trace.



FINDING SYSTEM NAMES

When working with SmartWizards you'll need to know the system names of the SmartObject properties that you're working with. To find these it is easiest to use the **View Xml** functionality in the SmartObject Service Tester tool to see the XML definition of the SmartObject that you need more information about. You can then copy this information from the tool into an XML editor, Visual Studio or Notepad++ to format it for easier reading. Also keep in mind that if you change the name of any of your entities after you've created your SmartWizards, then you must go back and change the system names in the layout.xml files.

If you can execute the SmartObject methods manually but the workflow events are failing, check to make sure that your system names are correct in your SmartWizards. For example, the following SmartObject mapping must use the system names for the SmartObjects, methods, properties and parameters. In the following example, the UpdatePet method has a parameter which is the serialized pet containing its properties. The system name of the parameter, which

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

you must use in the wizard, is **body_Update_UpdatePet** whereas the display name is **body (Pet) (1)**. The (1) is there because there's also the same parameter on the **AddPet** method.

```
<parameters>
    <parameter name="body_Update_UpdatePet" type="System.String" sotype="memo">
        <metadata>
            <display>
                <displayname>body (Pet) (1)</displayname>
                <description />
            </display>
            <service>
                <key name="typeFullName">PetStore.Pet</key>
                <key name="parameterInfoName">body</key>
            </service>
        </metadata>
        <mappings>
            <mapping type="parameter">
                <parameter name="body_Update_UpdatePet" />
            </mapping>
        </mappings>
    </parameter>
</parameters>
```

Tip

If you're tracing the network traffic using Fiddler, and when your workflow event executes the REST endpoint is never being called (e.g. in this case the /v2/pet API), it usually means that there is a problem with the system names not being correct. Use the View Xml menu in the SmartObject Services Tester tool to look at the definition of the SmartObject and compare each item in the SmartWizard definition with the actual generated system names of the SmartObject, methods, properties and parameters in question.

DEPLOYING SMARTWIZARDS

Once you are happy with your wizards you can copy folder under SmartWizards to each and every workflow designer's workstation. When you need to deploy the wizards to the thin client designers (K2 Designer and K2 for SharePoint), you must use the SourceCode.SmartWizards.RegistrationTool.exe tool located in the K2 blackpearl\Bin folder. This is detailed in the Help topic Custom K2 SmartWizards.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

BRINGING IT ALL TOGETHER

In summary, creating an end-to-end scenario based on a REST endpoint involves the following steps:

1. Creating and testing the descriptor file.
2. Creating an OAuth resource type and resource, if necessary, for use with the service instance.
3. Creating a service instance based on the descriptor file and generating SmartObjects (which are then used to test using a network tracing tool such as Fiddler to see the actual payloads).
4. Creating SmartWizards based on the SmartObject, being careful when specifying the system names of the SmartObjects and their associated methods, properties and parameters.
5. Creating and testing workflows that use the SmartWizards.

Using the strategy and tips found in this document, and the example attached to the download as well as the other samples in the product and online, you have a powerful way to include endpoint and line of business integration with K2.

APPENDIX A: CREATING A CUSTOM SERVICE CATEGORY

When you have multiple instances of a service type and you want to allow your workflow designers to choose an instance during design time, creating a custom service category is necessary. To do this use the following script, updating the five instances of the GUID with a new one and updating the path to the REST broker.

Note This requires that the REST broker is installed.

```
DECLARE @PetStoreService NVARCHAR(MAX);
SET @ PetStoreService = N'<servicetype name="PetStore" guid="f5c43eb6-669c-4336-a908-
aea02521f0cd">
  <metadata>
    <display>
      <displayname>PetStore</displayname>
      <description>Service Broker for PetStore</description>
    </display>
  </metadata>
  <config>
    <assembly path="C:\Program Files (x86)\K2
blackpearl\ServiceBroker\SourceCode.SmartObjects.Services.Endpoints.Rest.dll"
class="SourceCode.SmartObjects.Services.Endpoints.Rest.Broker" />
  </config>
</servicetype>';
IF EXISTS (SELECT 1 FROM [SmartBroker].[ServiceType] WHERE ServiceTypeGUID =
N'f5c43eb6-669c-4336-a908-aea02521f0cd')
BEGIN
       UPDATE [SmartBroker].[ServiceType]
```

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.

```
        SET             ServiceTypeXML = @LinkedInService
                        , ServiceTypeGUID = N'f5c43eb6-669c-4336-a908-aea02521f0cd'
        WHERE   ServiceTypeGUID = N'f5c43eb6-669c-4336-a908-aea02521f0cd'
END
ELSE
BEGIN
        INSERT ServiceType
                        ([ServiceTypeXML]
                        ,[ServiceTypeGUID])
        VALUES (@PetStoreService
                        ,N'f5c43eb6-669c-4336-a908-aea02521f0cd')

END
```

After running this SQL script against the K2 database, you will see a **PetStore** service type category and then when you Generate SmartObjects a new **PetStore** SmartObject category is created. If you do this multiple times for different types of REST-based service brokers, remember to change the GUIDs in the script for each new service.

© 2013 SOURCECODE TECHNOLOGY HOLDINGS, INC.