Web API

User's Guide

Summary

Chapter 1

Web API

1.1 Introduction

One of the most common ways of marketing software is through SaaS (Software as a Service), allowing complete usability via the web and installation in the cloud.

Applications are made available over the web through the use of a Web API (Application Programming Interface), which is an interface exposed on the web that allows exchanging data between a remote application and any client.

The architectural model commonly used in Web APIs is the RESTful model. It is based on the following fundamental principles:

- The application state and features are represented as WEB resources.
- Each resource is unique and addressable using a universal syntax for hypertext links.
- All resources are shared as a uniform interface for the transfer of state between client and resource. This consists of:
  - A bounded set of well-defined operations
  - A bounded set of content, optionally supported by code-on-demand
  - A protocol that is Client-Server, Stateless, Cacheable, and layered.

1.2 How to create Web APIs with Instant Developer

Creating Web APIs with Instant Developer is very simple. For each resource that you want to expose, you can simply create a DO class with the necessary properties and enable the *WebAPI* service. This automatically provides the following features:

- Reading, inserting, deleting, and updating a resource given its identifier
- Searching through a collection of resources given search criteria based on their properties

You can also expose specific features by creating class methods and enabling the *WebAPI* flag.

For example, suppose you want to create a Web API for querying products identified by an alphanumeric code (such as P10). First, you would create the Product class by dragging the corresponding database table onto the application. Then, simply enable the *WebAPI* service and compile the application, and information for the product with the code P10 can be retrieved by typing the URL *http://mydomain/myapp/Product/P10* in the browser.

1.3 Web API specification

Let's take a closer look at how to call a Web API created with Instant Developer.

1.3.1 Stateless calls

Calls are stateless. In other words, each request has no memory of the past and therefore the session is not managed.
If you want to use a non-stateless management mode, you can identify calls with a token and store the session information in a database or any other data source.

1.3.2 Data format

The default format of the data exchanged is JSON, but you can specify the desired format (JSON or XML) in the URL.

If you wanted to retrieve the product P10 in XML format, you would use the URL *http://mydomain/myapp/Product.xml/P10*.

1.3.3 Composition of URLs

The URL of Web API calls includes the following parts:

- The application path (e.g., *http://mydomain/myapp*).
- The name of the class (Tag) including any component package or namespace. For example, if the Product class is defined in a component with the namespace *com.progamma.catalog*, the URL should be followed by */com/progamma/catalog/Product*.
- The desired format (optional), specified by appending the string *.json* or *.xml* to the class name.

- Primary key (PK) values separated by forward slashes for read or delete requests (e.g. */P10/2010*).
- Search criteria for search requests or property values for update requests (e.g. *?Year=2010&Supplier=2*). Properties are identified by the corresponding value of Tag.
- Parameter values for custom method requests.

1.3.4 Types of calls

The types of calls that can be made depend on the HTTP method used as shown in the following table:

HTTP method	Type of call
GET	Reading a resource with the PK values in the URL.
GET	Searching for resources with the search criteria in the URL.
PUT	Updating an existing resource identified by the PK values in the URL, with the modified values in the URL or in the content.
POST	Inserting a new resource with the serialization of the resource in the content.
DELETE	Deleting an existing resource identified by the PK values in the URL.
METHODNAME	Call to a custom method with the parameters in the URL or in the content.

1.3.5 Depth of child object hierarchy

When making calls using the default Web API service functions (GET, PUT, POST, and DELETE), you can specify the level of depth for child objects to include by setting a request header named *child-level*.

If not specified, the default value is 0 for a GET with search criteria, and 9999 in all other cases.

1.3.6 Custom methods and call limitations

For calls to custom methods, the name of the method to be called must be specified as the HTTP method of the request. However, there may be situations where a call using a non-standard HTTP method is not possible, for example when:

- The language used on the client side to make the call does not allow it
- The firewall on the client or server does not allow it
- The web server does not allow it

In these cases, you can use POST as the HTTP method and specify the name of the method to be called in a request header named *X-HTTP-Override-Method*. The server checks if this header is present before deciding which operation to perform.

1.3.7 Encoding of values

The values exchanged in Web API calls must comply with the encodings listed in the following table:

Data type	Encoding
Null	Null values are encoded with the string *~~NULL~~* to distinguish them from empty string values.
Datetime	The format used for dates is *yyyy-mm-dd hh:nn.ss*.
Number with decimal point	For numeric values with a decimal point the decimal separator is used and the thousands separator is omitted.
Boolean	Boolean values are represented by -1 and 0.
BLOB	Blobs are represented in web format (e.g. etc.).
DocID	DocID values are encoded in the form of a GUID.
IDDocument and IDCollection	Only instances of these classes are serialized into XML or JSON. Use of any other type of object is reported by Instant Developer during the validation phase as not permitted for Web API calls.

Properties in XML, JSON and search criteria are identified by the corresponding value of Tag. These encodings must be used by the client for the data sent and are used by the server for the data returned.

1.4 Examples of calls

This section shows some examples of calls, one for each type. The examples call a hypothetical application reachable through the URL `http://www.progamma.com/NewWebApp`.

1.4.1 Example of reading resources

To retrieve a resource identified by key values (e.g., Invoice with Year and Number as a PK), you would make a call like this:

```
GET http://www.progamma.com/NewWebApp/Invoice/2013/1 HTTP/1.1
```

The response might be:

```
HTTP/1.1 200 OK
Content-type: application/json

{
  Year : 2013,
  Number : 1,
  ...
  Row : [ { id: 1, ... }, { id: 2, ... }, ... ]
}
```

Note that the rows of the invoice were also returned since the child-level header was omitted (thus defaulting to 9999), and the response is in the default JSON format since no format was specified.

1.4.2 Example of searching for resources

To retrieve resources that match certain search criteria (e.g. invoices for the years 2013 and 2014 with the status P), you would make a call like this:

```
GET
http://www.progamma.com/NewWebApp/Invoice.xml?Year=2013;2014&Status=P
HTTP/1.1
```

The response might be:

```
HTTP/1.1 200 OK
Content-type: application/xml; charset=utf-8
```

```
<IDCollection>
  <Invoice Year="2013" Number="1"  ... />
  <Invoice Year="2013" Number="2"  ... />
  <Invoice Year="2013" Number="3"  ... />
</IDCollection>
```

Note that the search criteria are specified in the URL query string using QBE syntax (see 2013;2014). In this case, the XML format was specified in the request URL, so the response is in XML format. Remember that for a GET with search criteria, the default value for the *child-level* header is 0, and since we omitted it in this case no invoice rows were returned.

1.4.3 Example of inserting resources

To insert a new resource (such as an invoice), you would make a call with the resource data in JSON or XML format in the content:

```
POST http://www.progamma.com/NewWebApp/Invoice HTTP/1.1
Content-type: application/json

{
  Year : 2013,
  Number : 1,
  ...
  Row : [ { id: 1, ... }, { id: 2, ... }, ... ]
}
```

If the insertion succeeds, the response will be something like this:

```
HTTP/1.1 204 OK
```

If the class in question has a counter field as the primary key, the response will include the value assigned by the counter:

```
HTTP/1.1 200 OK
Content-type: text/plain; charset=utf-8
10231
```

If the request fails, the response will be something like this:

```
HTTP/1.1 500 Internal Server Error
Content-type: text/plain; charset=utf-8

<Error messages>
```

1.4.4 Example of deleting resources

To delete a resource (e.g., the invoice from 2013 with the number 1), you would make a call like this:

```
DELETE http://www.progamma.com/NewWebApp/Invoice/2013/1 HTTP/1.1
```

If the request succeeds, the response will be similar to the response when inserting. If the invoice is not found, the response will be something like this:

```
HTTP/1.1 404 Not found
Content-type: text/plain; charset=utf-8

Invoice not found
```

1.4.5 Example of updating resources

To update a resource (e.g., invoice number 1 of 2013), there are two possible ways:

- Pass the values to be modified in the URL, in which case the call will be something like:

```
PUT     http://www.progamma.com/NewWebApp/Invoice/2013/1?Status=P&...
HTTP/1.1
```

- Pass the invoice data to be modified and the PK values in JSON or XML format in the content if you also want to update the child objects (or simply if you prefer this way):

```
PUT http://www.progamma.com/NewWebApp/Invoice HTTP/1.1
Content-type: application/json

{
  Year : 2013,
  Number : 1,
  Status : "P"
  ...
  Row : [ { id: 1, ... }, { id: 2, ... }, ... ]
}
```

The response will be like that of the previous cases.

1.4.6 Example of a call to a static method

To call a static method, you would make a call like this:

- If the method has no parameters or has simple parameters (such as the count of invoices for a year) the values of the parameters should be passed in the URL:

```
CALCULATE         http://www.progamma.com/NewWebApp/Invoice?Year=2013
HTTP/1.1
```

- If instead the method has at least one object-type parameter (e.g., IDCollection or IDDocument) the values of the parameters should be passed in the content:

```
POST http://www.progamma.com/NewWebApp/Invoice HTTP/1.1
X-HTTP-Override-Method: CALCULATE
Content-type: application/json

{
  Year : 2013,
  Number : 1,
  Status : "P"
  ...
}
```

If the method returns a simple value, the response will be something like this:

```
HTTP/1.1 200 OK
Content-type: text/plain; charset=utf-8

43
```

While if the method returns an object, the response will be like this:

```
HTTP/1.1 200 OK
Content-type: application/json

{
  Year : 2013,
  Number : 1,
  Status : "P"
  ...
}
```

Note that in the second type of call the method is specified in the *X-HTTP-Override-Method* header.

1.4.7 Example of a call to a non-static method

The following cases are examples of calling a non-static method:

- A resource identified by the PK in the URL and any simple parameters passed in the URL:

```
CALCULATEBALANCE
http://www.progamma.com/NewWebApp/Invoice/2013/1?Discount=0.5
HTTP/1.1
```

- A resource identified by the PK in the URL and object-type parameters passed in the content:

```
CALCULATEBALANCE     http://www.progamma.com/NewWebApp/Invoice/2013/1
HTTP/1.1
Content-type: application/json

{
  DiscountObj : { ... }
  ...
}
```

- A resource instance and object-type parameters passed in the content, in which case the instance must be identified by the name _ID_INSTANCE:

```
CALCULATEBALANCE http://www.progamma.com/NewWebApp/Invoice HTTP/1.1
Content-type: application/json

{
  _ID_INSTANCE:
  {
    Year : 2013,
    Number : 1,
    Status : "P"
    ...
  },
  DiscountObj : { ... }
  ...
}
```

The response will be the response for static methods.

1.5 Algorithm for handling a call

This section describes in detail how a Web API call is handled. First, it is worth noting that by enabling the WebAPI service on a class, in the application configuration file (*web.config* in C# and *web.xml* in Java) the necessary mappings are added to tell the Web server that the application also handles the specific Web API URLs.

When a call reaches the server, the service checks if it is a Web API call, which is the case if the URL does not end with the default document (AppName.aspx or AppName.htm). If it is not a WebAPI call, the request will be handled by the RD3 framework. In any case, the Initialize event is always raised, and it can be used to check the *WebApiService.IsWebApiRequest* function to determine the type of request.

Then, the name of the class and any format and PK values are extracted from the URL. If an instance of the class cannot be created, the call fails with the following error:

```
HTTP/1.1 405 Method Not Allowed
Content-type: text/plain; charset=utf-8

Class not found for uri '<URL path>'
```

The class is then checked to determine whether it has the WebAPI service enabled. If it is not enabled, the call fails with the following error:

```
HTTP/1.1 405 Method Not Allowed
Content-type: text/plain; charset=utf-8

Class '<class Tag>' not enabled for WebApi
```

After that, the method called is identified by checking the presence of the *X-HTTP-Override-Method* header. This is done using the *WebApiService.GetMethod* function. If it is not a basic method (GET, PUT, POST, or DELETE) the server checks to see if the WebAPI flag has been enabled for the method. If it is not enabled, the call fails with the following error:

```
HTTP/1.1 405 Method Not Allowed
Content-type: text/plain; charset=utf-8

Method '<method name>' of class '<class Tag>' not enabled for WebApi
```

If the class does not have a method with that name, the call fails with the error:

```
HTTP/1.1 400 Bad Request
Content-type: text/plain; charset=utf-8

Method '<method name>' of class '<class Tag>' not found
```

If it is a custom method, the server checks whether the method returns a serializable value, i.e., a simple type or an IDDocument or IDCollection object. Otherwise, the call fails with the following error:

```
HTTP/1.1 400 Bad Request
Content-type: text/plain; charset=utf-8

Method '<method name>' of class '<class Tag>' has a return value not serializable
```

The same check is also performed for parameters. If the check is unsuccessful, the call fails with the following error:

```
HTTP/1.1 400 Bad Request
Content-type: text/plain; charset=utf-8

Method '<method name>' of class '<class Tag>' has at least one parameter not serializable
```

Then, some additional checks are performed to see if the request conforms to specifications, for example:

- Parameters provided when not required or missing when required
- Parameters provided in the URL and also present in the content
- PK values provided when not required or missing when required
- Content provided when not required or missing when required

In all these cases, the response status code is always 400 Bad Request.

After that the content, if present, is read. If an error occurs during parsing, the call fails with the following error:

```
HTTP/1.1 400 Bad Request
Content-type: text/plain; charset=utf-8

Unable to parse content of the request. Invalid <format> format: <content>
```

Any parameters and/or the resource instance are extracted from the content. The parameters are converted to the correct type and the instance is loaded from the correct format. If an error occurs, the call fails with the status code 400 Bad Request and the text of the error.

Then, the *WebApiService.ChildLevel* property is initialized to 0 for resource searches or 9999 in all other cases. If the *child-level* header is present in the request, the property is set with the value of the header.

After that, if there are PK values, the server checks whether their number corresponds to the number of PK values for the class. Otherwise, the call fails with the following error:

```
HTTP/1.1 400 Bad Request
Content-type: text/plain; charset=utf-8

Primary key mismatch: the class '<class Tag>' has <number of PK values
for the class> properties in the primary key, but the values passed are
<number of values passed>
```

The values are converted to the corresponding types, as happens for parameters, and if any value is not convertible, the call fails with the following error:

```
HTTP/1.1 400 Bad Request
Content-type: text/plain; charset=utf-8
```

Value '<i-th value>' cannot be converted for property '<Tag of i-th property>' of the class '<class Tag>'

At this point, if the request is not a read or search for a resource (GET), the *LoadFromDB* method is called on the instance. If loading is unsuccessful, the call fails with the following error:

```
HTTP/1.1 404 Not Found
Content-type: text/plain; charset=utf-8

<document DNA> not found
```

Then, for static methods, the parameters are checked and converted. If an error occurs, the call fails with the status code 400 Bad Request and the text of the error.

For search requests, the search criteria are read. If a matching property is not found, the call fails with the following error:

```
HTTP/1.1 404 Not Found
Content-type: text/plain; charset=utf-8

Property '<criteria name>' not found"
```

For update requests with the values to be modified in the URL, the values are copied to the instance. The call fails with status code 400 Bad Request if a matching property is not found or if the value cannot be converted.

For update request with an instance in the content, the original instance is loaded, and, from the instance passed, the values are copied to the corresponding properties. If

the instance provided also has children, the same procedure is applied. If a child is not found in the original instance, it is added and set as inserted. The server does not check whether the instance provided is missing any children for possible deletion.

For insertion requests, documents are simply marked as inserted in the hierarchy to the level specified in the *WebApiService*. property.

At this point, the request headers are copied, and can be queried using the *WebApiService.GetHeaders* method.

Finally, the *OnWebApi* event is raised to the document instance. You can use this event to change the default operations performed by the framework up to that point (see next section). If the operation has not been canceled by setting the Cancel parameter to true, the framework proceeds with the operation and responds with the result.

If any other exception occurs throughout the process, the call fails with the following error:

```
HTTP/1.1 404 Not Found
Content-type: text/plain; charset=utf-8

Unknown error: <exception text>
```

In each case, except the one described below, the session is terminated.

1.6 Customization code examples

This section shows some examples of using the *OnWebApi* event to change the default behavior of the framework when handling calls to a Web API method.

1.6.1 Authentication using headers

If you want to handle authentication using headers, you can globalize the *OnWebApi* event and handle it as follows:

```
event MyDocHelper.GlobalDocumentWebAPI(
  IDDocument Document  // Source Document
  inout boolean Cancel // A boolean output parameter. If set to true it st...
)
{
  // Read username header
  IDMap requestHeaders = WebApiService.getHeaders()
  string username = requestHeaders.getValue("username")
  //
  // Find employee with this username
  int vIdEmployee = 0
  if (username != null)
  {
    select into variables (found variable)
      set vIdEmployee = EmployeeId
    from
      Employees // master table
    where
      LastName = username
  }
  //
  // Employee not authorized; abort the request
  if (vIdEmployee == 0)
  {
    Cancel = true
    WebApiService.setResponse(formatMessage("User |1 not found", username, ..
        ), 401, ...)
  }
}
```

Example loading using headers

The *WebApiService.GetHeaders* function is used in the event to retrieve the headers received in the request. The call is also canceled by setting the Cancel parameter to true.

When a call is canceled, the framework must be told what to write in the response using the SetResponse method.

In this case the response is as follows:

```
HTTP/1.1 401 Unauthorized

User <username> not found
```

1.6.2 Filtering properties

When enabling the WebAPI service on an existing class, you may want to hide some properties to prevent them from being listed in the response.

Suppose, for example, you do not want to show the supplier of the Order class. To do this, simply handle the *OnSaveXMLProp* event and clear the ExternalTag parameter.

```
event Order.OnSaveXMLProp(
  string InternalName      // String parameter. The name of the property o...
  inout string ExternalTag // String output parameter The name that the sy...
  inout string Value       // The value of the property that will be saved...
  inout boolean UseElement // A boolean output parameter. Tells the system...
)
{
  // Only for WebApi request
  if (WebApiService.isWebApiRequest())
  {
    // hide shipper property
    if (InternalName == "SHIPPER")
      ExternalTag = ""
  }
}
```

Example of filtering properties

The property with the SHIPPER Tag will not appear in the response.

1.6.3 Filtering child objects

You may also want to hide some child objects in the hierarchy. For example, suppose you have an Order class with two collections, OrderDetails and OtherCollection, and do not want to show the children contained in OtherCollection. To do this, you can simply handle the *OnWebApi* event and set the Loaded property for the collection to true.

```
event Order.OnWebAPI(
  inout boolean Cancel // A boolean output parameter. If set to true it st...
)
{
  // Only for request of search and load
  if (WebApiService.getMethod() == "GET")
  {
    // Set OtherCollection loaded
    OtherCollection.loaded = true
  }
}
```

Example of filtering children

After the event is fired, the WebAPI service loads the child objects, and when it finds the collection's loaded property set to true, it skips loading the collection. The child objects of that collection will therefore not be shown in the response.

1.6.4 Custom search

You may also want to respond to searches in a customized way.

For example, suppose that for a search of orders you want to respond with an empty collection. To do this, you can simply populate the *WebApiService.OutputCollection* property.

```
event Order.OnWebAPI(
  inout boolean Cancel // A boolean output parameter. If set to true it st...
)
{
  IDMap parameters = WebApiService.getParameters()
  //
  // Only for request of search
  if (WebApiService.getMethod() == "GET" && parameters.length() > 0)
  {
    // Return an empty collection
    IDCollection empty of Order = new()
    empty.loaded = true
    WebApiService.outputCollection = empty
  }
}
```

Example of a custom search

After the event is fired, the WebAPI service detects that the collection has already been loaded and does not try to load it. The response will therefore contain an empty collection.

```
HTTP/1.1 200 OK
Content-type: text/xml; charset=utf-8

<IDCollection/>
```

1.6.5 Deep linking

The WebAPI service can also be used to obtain Deep linking functionality. For example, suppose you want to be able to type the URL

http://www.progamma.com/CRM/Order/43 to open the CRM application and directly show the orders form on the order with the key 43. To do this, you can simply handle the *OnWebApi* event as follows:

```
event Order.OnWebAPI(
  inout boolean Cancel // A boolean output parameter. If set to true it st...
)
{
  if (WebApiService.getMethod() == "GET")
  {
    // Ser userRole to bypass login page
    CRM.userRole = Administrator
    //
    // Show the desired form
    Orders.show([OpenAs])
  }
}
```

Example of deep linking

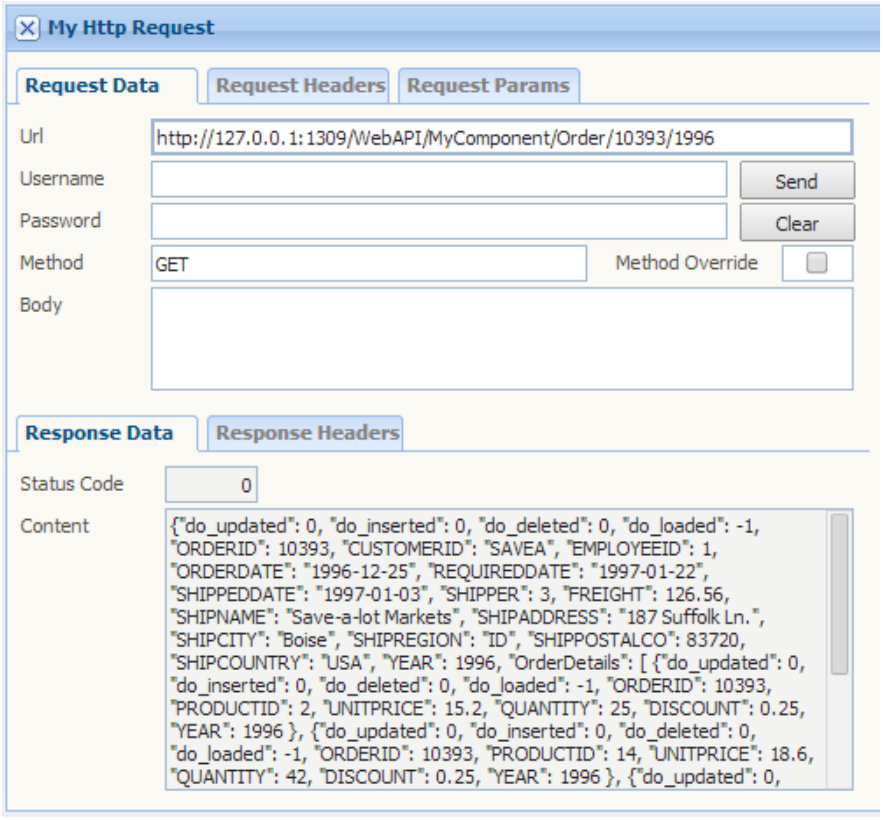
After the event is fired, if the WebAPI service detects that the SetResponse method has not been called, it performs a redirect to the default document causing the application to respond with the user interface. It is important to remember to manage the UserRole property, because otherwise the login form would be displayed.

1.7 Testing a Web API

Web API classes can be tested using HTTP protocol, so you can use any client that can make HTTP requests.

However, we have also made available a Web application, created with Instant Developer, that is especially useful for testing Web API applications. It can be downloaded from this link. The application contains a single form. The top part of the form is used to enter the request data (URL, method, header, parameters, etc.). The request is sent by clicking the Send button.

The response data is displayed in the bottom part of the form.



WebAPI test form

If the value returned is not what you expect, you can compile the server with file debugging to see how the request was handled on the server side.

Chapter 2

Integration with RESTful Web API services

2.1 Introduction

InDe makes it possible to integrate RESTful services into your applications. There is a convenient wizard for creating DO classes in the project corresponding to the entities of the service. At runtime, these classes interact with the service to retrieve the necessary data and perform CRUD operations. You can simply create forms based on these classes for displaying and editing the resources.

2.2 Importing a service

To integrate a service, you first create DO classes in the project corresponding to the entities to be accessed. To do this, open the project in InDe and launch the *Web API Importer* wizard from the *Tools* menu.

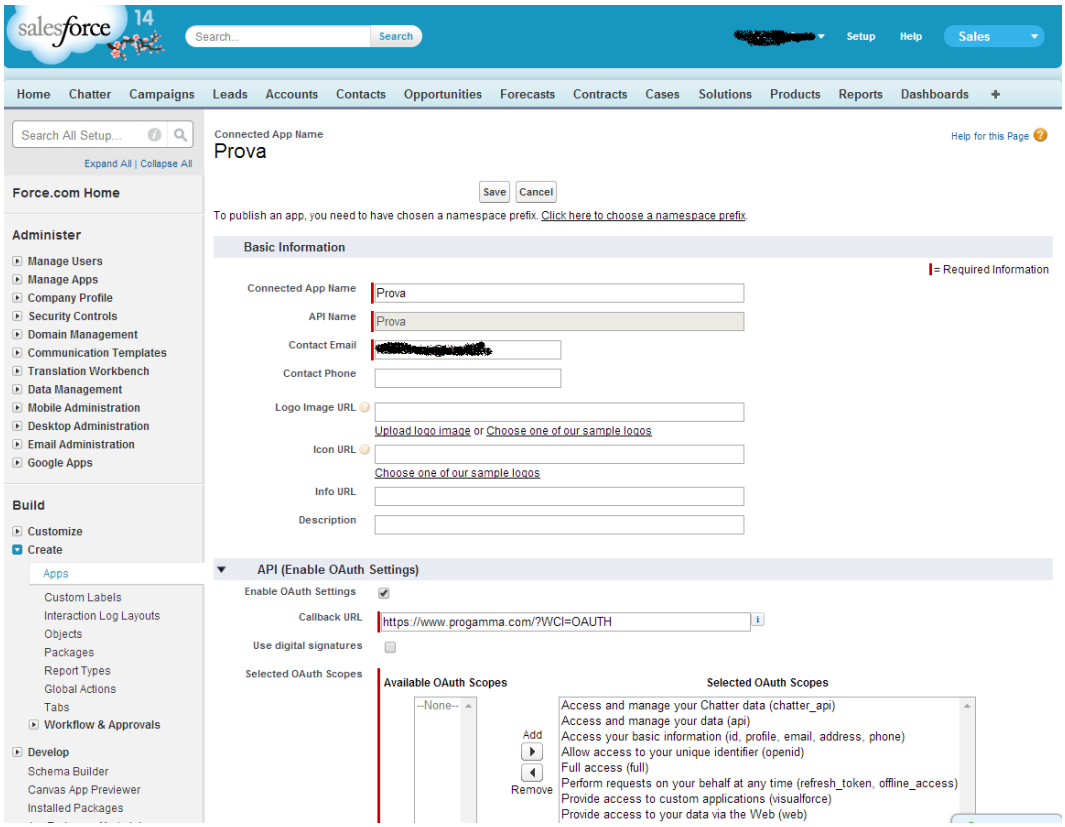
The wizard has an option for Salesforce and OData as well as a generic one, and if necessary it can manage OAuth2 and Basic authentication.

Wizard for importing RESTful Web API services

2.2.1 Authentication

To maintain secure communication, many services require OAuth2 authentication, and you normally have to register applications from which you want to access such services. To authenticate using the wizard, you have to register the url *https://www.progamma.com/?WCI=OAUTH*.

On Salesforce, applications are registered in the section *Setup / Build / Create / Apps* as shown in the image below.



Registering an application on Salesforce

Once the application is registered, you are provided with an ID and a secret key that must be entered in the respective *ClientID* and *ClientSecret* fields in the wizard. You must also specify the endpoint and the access token. In the case of integration with Salesforce, these two items are automatically set.

Once the data is populated, you only have to click the *Request Token* button to start the authentication and obtain a valid access token for communicating with the service.

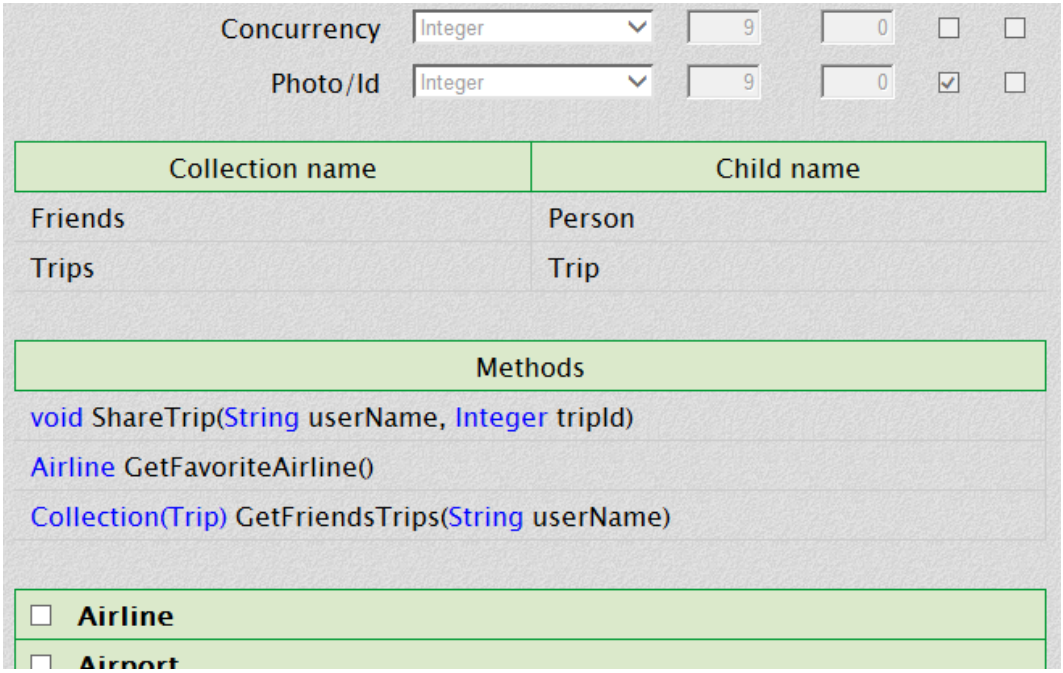
2.2.2 Salesforce

After authenticating with Salesforce, click the *Import* button in the *Classes* section of the wizard. The list of entities exposed by Salesforce will be populated. During the operation, you can filter the classes to be imported by using the search field shown in the image below.

Search field for Salesforce classes

2.2.3 OData

For OData services, as with Salesforce, you can simply click the *Import* button after specifying the url of the service to retrieve the list of entities exposed by the service.
The wizard can also import the Actions and Functions exposed by OData services, which will be listed below the collection in a specific section.



Imported Actions and Functions of OData services

2.2.4 Generic service

For services other than Salesforce, the list of exposed entities is not available, so you have to import one class at a time. To do this, you can specify the url of a resource or directly provide the JSON or XML code representing it. In the first case, when the Import button is clicked, the wizard will contact the service to obtain the corresponding JSON or XML.

If the service requires particular headers for requests, you can enter them in the corresponding field, one per row.

When you click the button, the wizard calculates the endpoint of the service. If the endpoint is incorrect, it must be changed before creating the class in the project.

Based on the values read, the wizard automatically determines the data type of the various properties. If the value of a property is null, its type must be specified explicitly.

2.2.5 Creating classes

Whatever the type of service, you can see the structure of a class by clicking on its name. This will show the list of properties, collections, and methods (only for OData).
At this point, simply select the desired classes and click the Create button, and the corresponding DO classes will be added to the project. During creation of each class, collections and methods will be skipped that refer to non-imported classes.

2.3 Integration with a service at runtime

After adding the classes for a service to the project, you can use them like normal DO classes in panel master queries and can display and edit the data of the related entities.

Using a map, at runtime each service is associated with an instance of the IOTConnector class, which serves as the connector function and takes care of making HTTP calls to the service.

Through the ServiceEndpoint property, each imported class knows which service to reference, contacting the service automatically whenever its data is to be read or modified.

The map of connectors is populated automatically by the framework based on the information stored in each class during the import using the wizard. To retrieve or associate a connector, you can use the GetServiceConnector and SetServiceConnector methods.

For services that require OAuth2 authentication, the AuthorizationEndpoint, TokenEndpoint, ClientID, and ClientSecret properties must be set to manage the authentication.

For services that require Basic authentication, meanwhile, the Username and Password properties must be set.

For both the types of authentication, when the corresponding properties are set, the framework automatically adds the *Authorization* headers to each request sent to the service using the credentials provided.

2.3.1 OAuth2 authentication

The IOTConnector class manages OAuth2 authentication for services that use this specification.

As with the wizard, you first have to register the application on the website of the service. In this case the url to be registered must match the one set in the RedirectUrl property, initialized to the application url and appended with the default document,

followed by the WCI = OAUTH parameter (e.g. https://mydomain/myapp/myapp.aspx?WCI=OAUTH). Salesforce requires, as recommended for the specification, that the url be secure, using the https protocol.

OAuth2 authentication involves obtaining an access token from the service that identifies the user, to be sent as a request header.

The access token is obtained through a specific sequence of interactions with the service, executed by calling the Authenticate method.
It starts with a redirect to the url specified in the AuthorizationEndpoint property to obtain an authentication code with the following parameters:

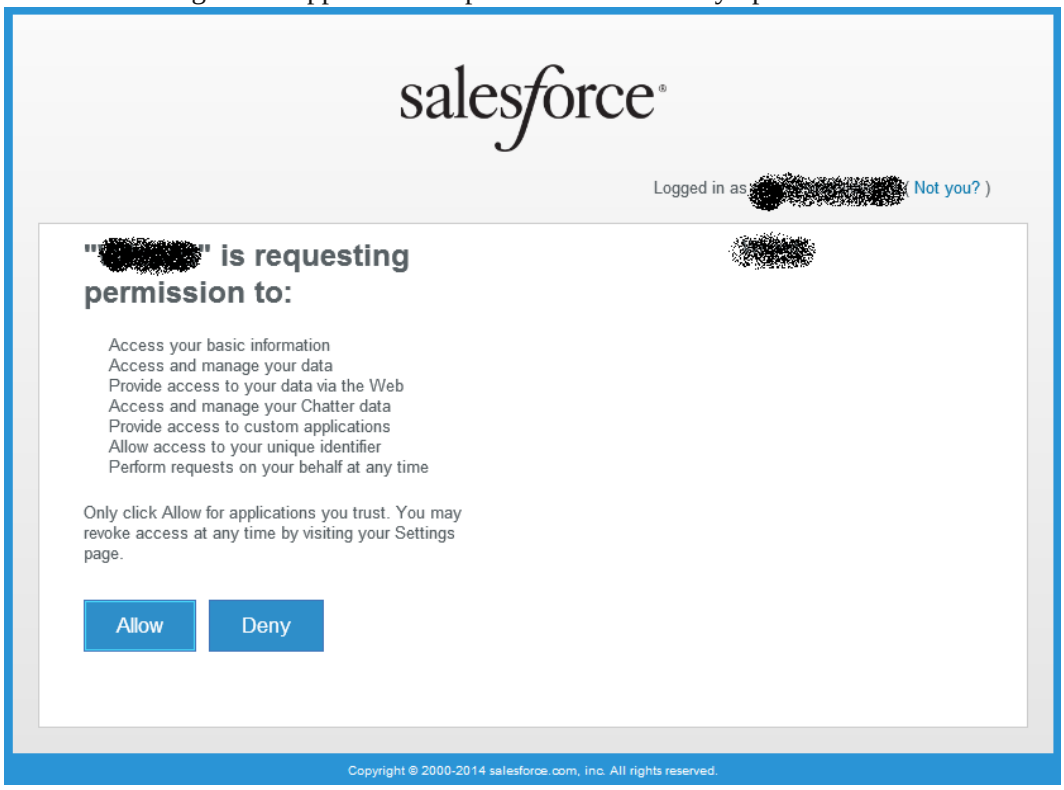
- client_id = ClientID
- redirect_uri = RedirectUrl+ "?WCI=OAUTH"
- response_type = code
- state = base64.encode(ServiceEndpoint)
- scope = Scope (for services that require it)

Before executing the redirect, the BeforeOAuthRequestCode event is raised, which you can use to add and/or modify the request parameters set by the framework if necessary for a particular service.

After the redirect, the user will be presented with a login page for the service, requiring a username and password to be entered.

Salesforce login page

When the correct data is entered, the user will see a second page prompting to explicitly authorize the registered application to perform the necessary operations.



Salesforce application authorization page

Whether the user gives the authorization or declines it, the system will be directed to the url specified in the RedirectUrl property. The framework recontacts the service decoding the endpoint from the state parameter, retrieves the connector from the map, and fires the AfterOAuthRequestCode event. The request should contain the code parameter, unless an error has occurred. If there is an error, the procedure stops. Otherwise it continues, composing a request to the url specified in the TokenEndpoint property to obtain the access token with the following parameters:

- code = code extracted by the parameters of the previous request
- grant_type = authorization_code
- client_id = ClientID
- client_secret = ClientSecret
- redirect_uri = RedirectUrl+ "?WCI=OAUTH

Before executing the new call, the BeforeOAuthRequestCode event is raised, which you can use to add and/or modify the request parameters set by the framework if necessary for a particular service.
After the call, the AfterOAuthRequestToken event is raised. From this point, if successful, the application will have the AccessToken needed to communicate with the service.

In addition to the AccessToken, the response parameters will contain the RefreshToken used by the framework to automatically update the AccessToken when it expires. It's best to save the AccessToken, RefreshToken, and AccessTokenLifetime parameters to be able to reinitialize them at the beginning of each session, thus avoiding the need to reauthenticate the user every time.

2.3.2 Communicating with the service at runtime

To display and edit the data for a service, simply create panels, trees, and books based on the DO classes created using the wizard. The framework will retrieve the connector associated with the service and will use it to manage the communication.

The service classes, in addition to serving as data sources for panels, can also be used in lookup, SmartLookup, and value source queries.

To manipulate the data for the service from code, simply call the usual functions, namely:

- LoadFromDB to load a resource
- LoadCollectionByExample and LoadCollectionFromDB to load a collection of resources
- SaveToDB to insert, update, and delete a resource

For generic services, the framework can only manage loading of a resource using the primary key and the insert, update, and delete operations. The following table shows which http method is used for the different operations:

Operation	HTTP method
Loading	GET
Insert	PATCH (with tunneling)
Update	POST
Delete	DELETE

For each operation, a specific event is fired by the connector similar to that of DO classes.

Operation	HTTP method
Loading a resource	BeforeLoad
Loading a collection	BeforeLoadCollection
Insert, update, delete	BeforeSave
Smartlookup	OnGetSmartLookup
Value source	OnGetValueSource

To act on calls made by the connector, you can create a class that extends the IOTConnector class and implement the above events, keeping in mind that after the event is fired, the framework performs the calls with the parameters of the event without recalculating them from the properties and the status of the document.