# Progress Data Object Handler Guide

Building back-end web services with the Data Object Handler

September 2018

Guide Contents

[Summary 3](#_Toc525216593)

[Prerequisites 3](#_Toc525216594)

[Creating a PAS Instance 4](#_Toc525216595)

[Creating a Sample Project 4](#_Toc525216596)

[Creating an ABL Service 5](#_Toc525216597)

[Service Naming 5](#_Toc525216598)

[Exploring the Internal Handlers 6](#_Toc525216599)

[Void Handler Example 7](#_Toc525216600)

[File Handler Example 9](#_Toc525216601)

[Entity Handler Example 10](#_Toc525216602)

[Extended Entity Options 12](#_Toc525216603)

[Using DOH Class Events 14](#_Toc525216604)

[Integrating with a CCS Application 18](#_Toc525216605)

[Integrating with the Progress Spark Toolkit 19](#_Toc525216606)

# Summary

The **Data Object Handler** (simplified as **DOH**) is a built-in class which implments the **IWebHandler** interface. This class is available in the **Progress Application Server for OpenEdge (PASOE)** via the **WEB** transport and provides the basis for accessing HTTP request and response payloads. Think of it as a more dynamic and business-ready form of a WebHandler, where the logic for parsing URL paths and HTTP verbs will be parsed via configuration file(s). Once information is parsed, any target ABL classes will be dynamically invoked and the appropriate class method will be executed. In addition to this configurable operation, since the DOH is an ABL class there are extension points through class events and overriding of certain internal handler classes.

The DOH has a known and traditional use for exposing the **Progress Data Object** pattern via annotated Business Entities in a somewhat automatic manner. But what about an API-first use-case where you want to design your own service interface with a custom URL pattern? Previously the only way to accomplish this was by use of a Mapped-REST service—basically an **ABL Service** which uses the REST transport but does not follow the Data Object Service pattern. In this case all HTTP mappings for the REST service would be done via the mapping GUI—a useful tool, but limited in functionality and options for exposing all possible RESTful behaviors. Conversely, a DOH-based WEB service offers a tremendous amount of flexibility limited only by the currently-available DOH class. In the following guide we will learn how to create our own custom service using the DOH service-mapping pattern.

# Prerequisites

To provide the best experience with **PASOE** and the **Progress Developer Studio (PDSOE)** it is recommended that you be on the latest service pack of OpenEdge. Use of OpenEdge 11.7.3 at minimum is preferable due to significant bug fixes and improvements around the DOH and its related classes. Therefore, use of a **64-bit Windows** installation and **OE 11.7.3 or later** is assumed, as well as the presence of the **PDSOE** component.

# Creating a PAS Instance

Before we begin creating a project it would be beneficial to have a valid PASOE instance available for deployment. Begin by **running** the command below and **adding** the server to your PDSOE environment (in the Servers view) as just “**SamplePAS**”. Once created, start the instance and visit <http://localhost:8870> to confirm the instance is running as expected.

pasman create -v -f -p 8870 -P 8871 -j 8872 -s 8873 C:\PASOE\SamplePAS

# Creating a Sample Project

Use of the PDO pattern is typically seen while creating a new **ABL Web App** project for PASOE. We will begin a new project by following this pattern and slowly adapt it to expose our own customized services via the WEB transport. The following steps will create this new project.

1. File > New > **OpenEdge Project**
2. Name our new project “**Sample**”
3. Select **Server** and **PAS for Openedge** option
4. Select the **WEB** transport from the dropdown
5. Check the box for “**Create a Data Object Service**”, click **Next**
6. Deploy as a WebApp “**Sample**” (keeping the default name)
7. Select the new PAS server called “**SamplePAS**”
8. Click **Finish**
   1. This will use the default service name of /SampleService
   2. No database connections will be required for our examples.

In anticipation of some future examples we need to make a change to the WebApp to always seal the Client-Principal token from Tomcat and return the current user’s name—otherwise some of our future code will fail if no valid CP token exists. By default we are using the **Anonymous** security model and simply need to adjust the **PASOEContent/WEB-INF/oeablSecurity.properties** file with the following options.

OEClientPrincipalFilter.**anonymous**=true

OEClientPrincipalFilter.**sealAnonymous**=true

OEClientPrincipalFilter.**passthru**=true

**Save** the file after modifying, **publish** the recent changes, and **start (or restart)** the PAS instance to ensure these modifications to the security model are picked up.

# Creating an ABL Service

The purpose of an **ABL Service** is to create a transport-dependent grouping for resources within an ABL WebApp using a **single transport** (eg. you cannot mix WEB and REST in the same ABL Service). If we examine our new project, you should notice an initial **Defined Service** created called “**SampleService**” which was created as a **Data Object Service**. If you double-click on this service it will reveal a Sample URI which will answer any request at **/web/pdo/SampleService** as indicated by the special **/web/pdo** prefix and Service Relative URI of “/**SampleService**”. In this pattern the service would be further configured via PDSOE by simply choosing annotated Business Entity classes to expose. The tooling reads the annotations and produces a **.gen** (for “generated”) mapping file with the necessary metadata to execute the exposed ABL code.

**In the remaining sections we will explore how to configure the DOH manually.**

## Service Naming

When working with the DOH in a customized manner, the first element after the transport (eg. “web”) is what the class considers the “service name”. This is important to operation as the DOH class will be looking for a service mapping file (**.map**) by the same name as the service being accessed. We will now define a new ABL Service which uses the WEB transport, though in this case we will manually select the DOH class and define our own Resource URI entries (rather than use the default of “/pdo” for Data Objects). The following steps will create the necessary service for our remaining examples.

1. Right-click on the project, select **New > ABL Service**
2. Select the **WEB** Transport if not already selected by default
3. Use the Service Name “**HelloService**” (this will be used only by PDSOE)
4. **Browse** and select “**OpenEdge.Web.DataObject.DataObjectHandler**”
5. Add a single **Resource URI** of “**/HelloService**” to match our service name.
6. Click on **Finish** to create the ABL Service.
7. Expand the **SamplePAS** server view to confirm **HelloService** has been added.

Now we need a way to control the available operations of the service and allow it to respond with data. **We will do this by placing a mapping file in the application’s PROPATH.** Begin by creating a new text file “**HelloService.map**” in the project’s **PASOEContent/WEB-INF/openedge** folder. The filename of “**HelloService**” should allow it to “map” any rules to the service endpoint “**/HelloService**” which we just created. Placing it in the “**openedge**” folder makes it discoverable via the application **PROPATH**. This file will contain a JSON payload to describe the behavior of the service, and at a minimum must contain a single object with a property “**services**” as an object. Within the “**services**” is a child object named for each exposed service (eg. “**HelloService**”). The “**version**” property here defines the version of the service and should follow the Semantic Versioning pattern (eg. “**1.0.0**”). The following JSON snippet will provide enough to make the service discoverable but not do very much else. Without this minimum configuration the server would only return an HTTP-404 response for the service.

{

    "services": {

        "**HelloService**": {

            "version": "1.0.0"

        }

    }

}

# Exploring the Internal Handlers

The DOH provides 3 standard handlers for operation of configured endpoints: void, file, and entity. Their basic behavior is outlined as follows.

* For the **void** handler, a statusCode is returned and no further action is taken.
* For the **file** handler, the value of the property is a path (relative or absolute).
  + AVM Substitution parameters are supported, which must start with “$”.
  + eg. $oepas-webapp, the current WebApp used to service this request.
  + Any environment variable available via the ABL's OS-GETENV() function.
* For the **entity** handler, this will be used to return an object instance.
  + This may be created by a custom factory/service manager or a simple DYNAMIC-NEW will be used by the DOH if no overrides to the LoadEntity event successfully produced an entity instance (more on that later). Both classes and procedures are supported for exposure as entities.
  + This will call some method in the entity, passing any configured input or output parameters as arguments (configured in an “arg” array).
  + Note: If a return value is used, it will be available as a “\_retVal” arg property. For the initial implementation arrays are only supported for RETURN values, not parameter/argument values.

## Void Handler Example

First we will look at the Void handler, which is a no-op event when used (meaning it does not execute any custom ABL code). This handler simply returns the provides HTTP status code, which can be useful to designate an endpoint as present but perhaps not usable. To illustrate this in our first example, accessing the root of the HelloService at “/“ via GET should result in returning an HTTP-501 (Not Implemented) response code. Since there is no configuration for the Void handler we simply set the value of this property to null. **Update** the HelloService.map file as highlighted below, **save** the changes, and **publish** to the PAS instance.

{

    "services": {

        "HelloService": {

            "version": "1.0.0"**,**

**"operations": {**

**"/": {**

**"GET": {**

**"contentType": "application/json",**

**"statusCode": 501,**

**"void": null**

**}**

**}**

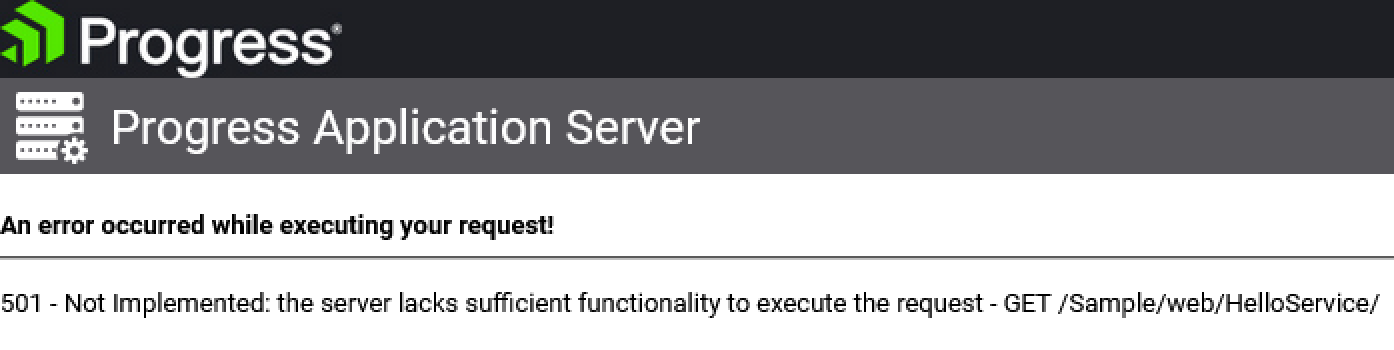
**}**

        }

    }

}

To test the changes, visit <http://localhost:8870/Sample/web/HelloService/> and confirm that you get an **HTTP-501** response from the server. **Note:** The resulting page may be shown as an error due to an unknown status code and/or response payload.



So how could this handler be useful? A few common reasons might be that you want to reserve a particular endpoint name for future enhancements. As was shown in our example, providing the mapping and returning an “HTTP/501 – Not Implemented” code would indicate that the endpoint name is valid but does not currently provide data. Another use could be that you wish to use the endpoint for confirming the server is alive, but with no data returned. Returning an “HTTP/204 – No Content” response could fulfill this example by stating the response was successful but has nothing further to say.

## File Handler Example

Next, we should create a more functional endpoint by use of the internal file handler. To begin, we should create a new plain-text file “**helloworld.json**” at **PASOEContent/static/** with the following JSON payload:

{

    "hello": "world"

}

Return to the **HelloService.map** file to add a new relative URI to our service. Simply update the file with the following configuration values for the “**/world**” endpoint and publish any changes to the PAS instance (if not performed automatically). This will utilze the internal File handler to return the “**helloworld.json**” file we just created.

{

"services": {

"HelloService": {

"version": "1.0.0",

"operations": {

"/": {

"GET": {

"contentType": "application/json",

"statusCode": 501,

"void": null

}

**},**

**"/world": {**

**"GET": {**

**"contentType": "application/json",**

**"statusCode": 200,**

**"file":**

**"$CATALINA\_BASE/webapps/$oepas-webapp/static/helloworld.json"**

**}**

**}**

}

}

}

}

If configured as expected, accessing the site at the following URL will result in seeing the contents of the new “**helloworld.json**” file. If so, congratulations! We now have the basis of a custom API and can begin creating some ABL code to return data.

<http://localhost:8870/Sample/web/HelloService/world>

## Entity Handler Example

Lastly, we’ll take a look at the **Entity** handler type. While the name may sound similar, this is NOT the same as a "Business Entity”. The latter is a very distinct type of business logic class which supports the CRUD+Submit operations as expected by the Progress Data Object (PDO) pattern. What we want here is a simple ABL class which will expose abitrary methods to the public via our mapped interface. We can create this new class via the following steps.

1. Right-click on the Sample project and select **New > ABL Class**
2. Set the Package Root as **Sample > PASOEContent/WEB-INF/openedge**
3. Set the Package name as “**Business**” to create the class in a subfolder
4. Name the class “**HelloUser**” and click **Finish**

With the new class created and opened automatically, right-click within the code and select **Source > Add Method**, set the method name to “**SayHello**” and return type left as VOID. Click the Generate button to add the method to our class. Add an **output** parameter “**greeting**” as **character** and then add the code below into the body of the method. This should return a greeting which uses the current user’s name, as provided by the Client-Principal object for the request.

define variable oPrincipal

as OpenEdge.Security.Principal no-undo.

assign oPrincipal = OpenEdge.Security.Principal

:Import(session:current-request-info).

assign greeting = substitute("Hello &1",

oPrincipal:Token:user-id).

delete object oPrincipal no-error.

To connect this new class and method to an interface, we need to again update our **HelloService.map** file. Add the entry below to the file just after the definition for the “**/world**” object. This should add a new endpoint “**/user**” to our service which will be handled by the Entity of “**Business.HelloUser**”. The internal block to be executed (via the property “function”) is our “**SayHello**” method which has an output argument of “**greeting**”. This is defined with the original ABL parameter name and datatype while being configured for output as a “**field**” of the response body. Once the file is updated and saved, **publish** all of the changes to the instance.

{

"services": {

"HelloService": {

"version": "1.0.0",

"operations": {

...

}**,**

**"/user": {**

**"GET": {**

**"contentType": "application/json",**

**"entity": {**

**"name": "Business.HelloUser",**

**"function": "SayHello",**

**"arg": [**

**{**

**"ablName": "greeting",**

**"ablType": "CHARACTER",**

**"ioMode": "OUTPUT",**

**"msgElem": {**

**"type": "field",**

**"name": "greeting"**

**}**

**}**

**]**

**}**

**}**

**}**

}

}

}

}

Once the PAS instance has completed publishing you will be able to test the new API at <http://localhost:8870/Sample/web/HelloService/user> which should respond with a JSON payload of **{“greeting”: “Hello anonymousUser”}**. If this matches your output then you now have working examples of the 3 available operation handlers.

## Extended Entity Options

In the initial example for the Entity Handler we let the DOH assume the type was “Class”, as we only provided a class package name and a method to be executed. If we add an additional property “**type**” we can further change some of the behavior of the entity and even extend this to execute procedural code! To begin, let’s create a procedure file with an internal procedure similar to our HelloUser class.

1. Right-click on the **Business** folder of your project, select **New > ABL Procedure**
2. Name the file/program “**HelloProc**” and click **Finish**
3. Add the following procedure at the end of the file and save your changes:

procedure SayHello:

define output parameter greeting as character no-undo.

define variable oPrincipal

as OpenEdge.Security.Principal no-undo.

assign oPrincipal = OpenEdge.Security.Principal

:Import(session:current-request-info).

assign greeting = substitute("Hello &1",

oPrincipal:Token:user-id).

delete object oPrincipal no-error.

end procedure.

1. In the **HelloService.map** file, add the following object **after** the “/user” definition added in the previous step. Remember to add a comma after the last object!
   1. Note the addition of the “**type**” property set to “**procedure**”
   2. Note the change in path style for the “**name**” property
   3. The remainder of the definition matches our class pattern, as we’re using the same name and datatype for the arguments (parameters).

**"/userproc**": {

"GET": {

"contentType": "application/json",

"entity": {

**"type": "procedure"**,

"name": "Business/HelloProc",

"function": "SayHello",

"arg": [

{

"ablName": "greeting",

"ablType": "CHARACTER",

"ioMode": "OUTPUT",

"msgElem": {

"type": "field",

"name": "greeting"

}

}

]

}

}

}

1. Save any changes and **Publish** to the PASOE instance (new .p, updated .map)

Once the PASOE contents have been republished, vist the new API at <http://localhost:8870/Sample/web/HelloService/userproc> to confirm operation. Both the class and procedure code are identical, so the response given by the new endpoint should match our previous “/user” endpoint example.

Another special value for the “type” property is the value “**self**” which implies that the answering DOH class will provide the function to be executed. However, this feature is currently limited to only select internal functions when used out-of-the-box with the default DOH class. As an example, the following will execute a special “**ListServices**” method within the default OE.W.DO.DOH class to return all available services in JSON format. Feel free to try this on your own using the processes outlined in previous examples. **Note: This is not the same format as the Data Object Service catalog.**

"**/list**": {

"GET": {

"contentType": "application/json",

"statusCode": 200,

"entity": {

**"type": "self"**,

"name": "**OpenEdge.Web.DataObject.DataObjectHandler**",

"function": "**ListServices**",

"arg": []

}

}

}

# Using DOH Class Events

Since the release of OpenEdge 11.6.3 there have been a few select events added to the DataObjectHandler class which are emitted throughout the lifecycle of a session or a request. As of 11.7.2 two more were added, so for the purpose of this section we will be referencing features available as of that version and later. To quickly summarize, the following events are available for subscription as necessary:

* **DiscoverService** – Called when a service is first referenced for use
* **LoadEntity** – When the entity handler must load the referenced class
* **Invoking** – Called just prior to executing the entity class method
* **Invoked** – Called just after executing the entity class method
* **OperationError** – Triggered whenever an error occurs within the entity
* **UnloadEntity** – Called when entity operation is complete (after invoked)

The subscription to the events must be done in a somewhat global manner to each session of each MSAS Agent, therefore we must have both a controlling class with our event subscriptions as well as a **sessionStartupProc** procedure to initially call our subscription class. This is easily accomplished through the following steps:

1. Use the **New > ABL Class** option to create a new empty class
2. Set the package root to **PASOEContent/WEB-INF/openedge**
3. Name the class “**DOHEventHandler**” and click **Finish**
   1. We will configure this in a future series of steps
4. Use the **New > ABL Procedure** option to create a new empty procedure
5. Set the container to **Sample/PASOEContent/WEB-INF/openedge**
6. Name the procedure “**startup.p**” and click **Finish**
   1. In the **startup.p** procedure add the following lines to the Main Block:

define input parameter startup-data as character no-undo.

new DOHEventHandler().

* 1. **Save** and **close** the file as we no longer need this

1. **Publish** all current changes to the PAS instance via PDSOE.
2. **Open** the file **C:\PASOE\SamplePAS\conf\openedge.properties** for editing
3. **Add** the following line to **[AppServer.Agent.SamplePAS]**:

sessionStartupProc=startup.p

1. **Save** the **openedge.properties** configuration file.
2. **Restart** the PAS instance.

Next, we will add subscriptions for some of the DOH events and examine what should happen with each event. For the remainder of our examples we only need to edit our **DOHEventHandler** class. At the top of the class, add the additional USING statement:

using OpenEdge.Web.DataObject.\*.

Add the following variable just after the class definition (the “class <name>” line):

define private variable oCheat as DOHEventHandler no-undo.

Add a **Constructor** and **Destructor** to the class, if not already present. We will use these to configure our subscribe/unsubscribe actions. Though before we proceed we need to make use of this “**oCheat**” variable we just created. Adding the following line to our **Constructor** method will create a reference of the class back to itself, thus “cheating” the garbage collection which might otherwise destroy this class.

assign oCheat = this-object.

Now we may begin to subscribe to the DataObjectHandler events. The basic syntax for this is as follows, and should be added to the **Constructor** for each available event as noted previously. For the “handler” portion, this should be the name of a method that exists in the current class (hence, this-object). The simplest way to name these is to follow the pattern of *<event> + “Handler”*, so the “**LoadEntity**” event should have a “**LoadEntityHandler**” method created in the class. We’ll perform that step shortly, though for now you can simply add each of the events with the names as suggested (and yes, line-wrapping as shown below should be fine).

OpenEdge.Web.DataObject.DataObjectHandler:<event>

:**Subscribe**(this-object:<handler>).

Similar to the subscription process, the unsubscribe works exactly the same way but with the “**Unsubscribe**” method. Ideally, for each ***subscribe*** you should have a matching ***unsubscribe*** in your class **Destructor**.

OpenEdge.Web.DataObject.DataObjectHandler:<event>

:**Unsubscribe**(this-object:<handler>).

To aid in quickly populating your class with the necessary methods, please find some sample stubs below complete with expected parameters. For the most part, each event takes only 2 parameters: a sender object and an event arguments object. The former is always a standard Progress.Lang.Object type, while the latter is typically a named class which partially takes the name of the event itself. Combined with the Subscribe/Unsubscribe actions in the Constructor/Destructor methods, this should complete the necessary setup for your class. **Note:** Properties in the event args may be modified at runtime by each method.

method private void DiscoverServiceHandler (

input poSender as Progress.Lang.Object,

input poEventArgs as ServiceDiscoveryEventArgs ):

end method. /\* DiscoverServiceHandler \*/

method private void LoadEntityHandler (

input poSender as Progress.Lang.Object,

input poEventArgs as HandlerLoadEntityEventArgs ):

end method. /\* LoadEntityHandler \*/

method private void InvokingHandler (

input poSender as Progress.Lang.Object,

input poEventArgs as OperationInvocationEventArgs ):

end method. /\* InvokingHandler \*/

method private void InvokedHandler (

input poSender as Progress.Lang.Object,

input poEventArgs as OperationInvocationEventArgs ):

end method. /\* InvokedHandler \*/

method private void OperationErrorHandler (

input poSender as Progress.Lang.Object,

input poEventArgs as HandlerErrorEventArgs ):

end method. /\* OperationErrorHandler \*/

method private void UnloadEntityHandler (

input poSender as Progress.Lang.Object,

input poEventArgs as HandlerLoadEntityEventArgs ):

end method. /\* UnloadEntityHandler \*/

To test the new events, simply try adding a message into the LoadEntityHandler method, publish the code, and attempt to run one of the DOH endpoints. The message should be immediately visible in the <app\_name>.agent.log file for the PAS instance.

With the events and their handlers in place you can now add your own custom logic. To assist with ideas, the following are merely ***suggestions*** as to how you can leverage these events in your own application:

* **DiscoverService** – Utilze the current URL path and version info (if configured for extraction) to locate your own custom service mapping. For instance, instead of utilizing the .map flat file this information could be stored in a database and retrieved dynamically. The basic process would include the following actions:
  + 1. Check the poEventArgs:Registry for a service with the given name/version.
    2. If not found, create a new DataObjectService instance for the name/version.
    3. Register the DOS instance via poEventArgs:Registry:RegisterService()
    4. Return the DOS instance via the poEventArgs:Service property
* **LoadEntity** – Utilize the CCS ServiceManager to start/obtain a service implementation for the necessary entity. A more detailed example will be given in a later section of this material.
* **Invoking** – Perform common processing rules prior to executing any services dynamically. For instance, the Spark Toolkit utilizes this point to establish any existing session context data.
* **Invoked** – Just after executing the necessary business logic, perform manipulations of the outbound data. For example, you could add additional parameters into a JSON payload, such as providing a count of records during a READ operation.
* **OperationError** – Detect and override the default error logic by providing your own custom HTTP status code/reason based on the type of error thrown by the application.
* **UnloadEntity** – After execution of logic, perform some custom cleanup of your application stack just as the entity class is being destroyed.

**Note:** For most of the above class events, so long as an error object is not assigned to the Error property of the arguments object, the default event logic will be executed as a fall-through. The only exceptions to this rule is if certain properties are set as a successful response, such as the Service property of the ServiceDiscoveryEventArgs object.

# Integrating with a CCS Application

As of the 11.7.2 release, the DOH class has been modified to test for existence of a ServiceManager class as part of a CCS implementation. (For reference, “CCS” is the “**Common Component Specification**” which describes a set of common manager classes which set the stage for a modernized application.) The available Progress Spark Toolkit from Progress utilizes this implementation pattern as a reference application and provides the necessary configuration options to work seamlessly with the DOH class.

In the case of the DOH implementation, during the **LoadEntityHandler** method the logic attempts to locate a valid entity implementation from the available **ServiceManager**. If present, the DOH uses the ServiceManager to start the entity class and return a valid class instance—otherwise the default logic will use the “dynamic new” feature to create a class instance.

As mentioned previously as part of the **LoadEntity** event, the handler for this event could be used to locate and start a custom entity rather than simply locating an available class in the PROPATH. This is beneficial to structured code as it allows for a single name of a class interface to be used as the target, while the ServiceManager takes care of starting (or reusing) the correct implementation of that interface.

The following code would be placed inside the **LoadEntityHandler** method created to deal with the subscription to the **LoadEntity** event. The code tests for a valid class as based on the operation’s target class name. If such a class exists, the ServiceManager from the CCS implementation is called upon to locate and either start or reuse the entity. Regardless of how an entity is obtained, it should always be assigned into the **BusinessEntity** property of the event arguments object.

/\* Attempt to load the requested service for this business entity. \*/

define variable oServiceClass as Progress.Lang.Class no-undo.

assign oServiceClass = Progress.Lang.Class

:GetClass(poEventArgs:Operation:TargetName)

no-error.

if valid-object(oServiceClass) then do:

/\* Start and obtain a service class instance from

the local CCS ServiceManager implementation. \*/

assign poEventArgs:BusinessEntity = Ccs.Common.Application:ServiceManager

:getService(oServiceClass).

/\* Leave a log message that the DOH event handler has been loaded. \*/

if valid-object(oLoggingManager) then

oLoggingManager:logMessage(substitute("Located Entity '&1'",

oServiceClass:TypeName), "DOH-LOAD", {&MIN\_LOGGING\_LEVEL}).

end. /\* valid-object \*/

# Integrating with the Progress Spark Toolkit

As part of the default [Spark-Toolkit](https://github.com/progress/Spark-Toolkit) repository and current release, the included code provides a **DOHEventHandler** class already configured to integrate directly with the DOH and its related ServiceRegistry classes. Whether using the PDO pattern or a customized DOH pattern, the included startup logic runs the event handler class and immediately registers any discoverable services for your application. Additionally, within the [Spark-Toolkit-Demos](https://github.com/progress/Spark-Toolkit-Demos) repository the included demo projects are configured to use service-level logging and illustrate examples of both the PDO and custom DOH patterns. The following differences for the available demos are outlined below:

* **Sports** – Follows the PDO pattern using the DOH, with .gen file produced by PDSOE and read from disk during the session startup. This approach requires use of annotated Business Entities and the PDSOE tooling.
* **DynSports** – Also follows a PDO pattern utilizing the DOH, but discovers available entities through use of the CatalogManager. The resulting metadata is then used by the ServiceRegistry to provide mapping information.
* **QuickStart** – Follows the DynSports approach using the DOH as a general API layer. No application code is provided aside from what comes within the Progress Spark Toolkit.