Appendix B

Progress Modernization Framework for OpenEdge

Customizing WebHandlers

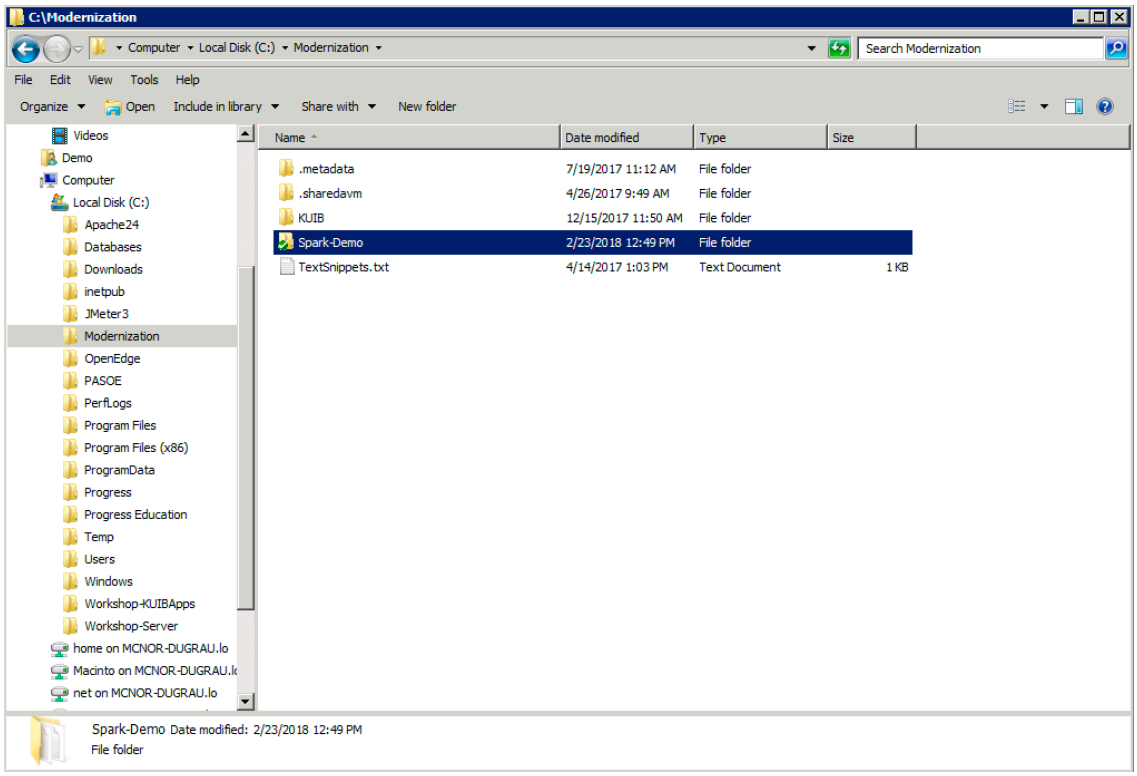
Purpose

This addendum provides supplemental information for configuring WebHandler classes in OpenEdge 11.6.3 and later. This specific version of OpenEdge is chosen due to its support for the DataObjectHandler, or DOH, which is the out-of-the-box WebHandler class that can provide metadata/catalog artifacts for exposing annotated BusinessEntity classes. The following material should illustrate a basic understanding of WebHandlers in general, though a specific focus will be on integrating the PMFO at a minimal level into a fully-custom WebHandler, or as part of normal DOH usage.

Obtaining Source Code

This operation assumes that you have installed an **Git** client, specifically **TortoiseGit**, to allow checkout of code from a GitHub repository. If using the VM provided for training, this should already be available.

1. Create a new directory to use as your PDSOE workspace, if not already available. The current practice has been to use **C:/Modernization** for consistency across various installations.
2. From the **C:/Modernization** directory right-click on an empty area within the directory and select the option "*Git Clone*". Enter the repository URL below:
   a. **https://github.com/progress/Spark-Demo**
   b. Be sure to set the destination directory to **C:/Modernization/Spark-Demo** and press **OK**.
3. You should now have a "*spark*" directory containing "*oe116*", "*oe117*", and "*support*".



NOTE

If the code has already been checked out, it is a good idea to make sure you are up to date with any recent changes before you begin working. This is easy with **TortoiseGit** and a copy of the repository. Just **right-click** on the new "*spark*" directory from within the Windows Explorer. Select the "*Git Sync*" option from the context menu, click on the "Pull" button, and any obtained modifications will be reflected in your directory.

Creating a WebHandler

When working with an ABLWebApp project, creating an OpenEdge.Web.WebHandler (or simply, WebHandler) is as direct as creating any new ABL class or procedure via the Right-click context menu in PDSOE. By default, the process for creating WebHandler does nothing spectacular, though it can optionally produce some stubs for the common GET, PUT, POST, and DELETE verbs of HTTP. Beyond that, you are responsible for adding any custom logic to the class to make it behave as you expect. Some example usage may include (but is certainly not limited to) the following:

- Accept binary data as part of a file upload endpoint.
- Return binary data as part of a [secure] file download.
- Accept arbitrary text data and parse for data import.
- Accept and return custom JSON-formatted data.

WebHandler Basics

WebHandler classes contain certain basic methods, some of which may be skipped and some which must always be present and overridden. When a new class is generated, it should contain at least an override for the **HandleNotAllowedMethod** and **HandleNotImplemented** methods. These may simply throw an error that state that they are not implemented. For purposes of this material we will not dive any further into these other than to say this override is sufficient for our purposes.

The key constructs within any WebHandler are a request object, a response object, and a response writer. For every standard HTTP verb is a handler method (HandleGet, HandlePut, etc.) that receives an **OpenEdge.Web.WebRequest** object as input. This class is essentially just a wrapper to the web-context handle and allows access to the raw HTTP data sent as request to the Tomcat server in PAS. There is no transformation of this data by the web server, and the entire request may be streamed in from the client thanks to the Web transport. From this request object, you have access to cookies, headers, parameters, the original URL, and of course a body payload (if supported by the HTTP verb). The extraction and use of this data will not be covered here, but can be found in several presentations by Peter Judge from various PUG Challenge events from 2014-2017. From the OpenEdge documentation portal, look up the **OEHttpClient** or view the interactive documentation here: https://documentation.progress.com/output/oehttpclient/oe117/index.html

Similarly, we cannot cover all the means to output data, though the most basic approach is to create an **OpenEdge.Web.WebResponse** object, set any custom cookies, headers, or body (called an entity in this object), and output that content via an **OpenEdge.Web.WebResponseWriter** which will flush the contents of the response back to the web server. The important item to remember is to make sure you catch any errors that may potentially arise during data processing within in a WebHandler, and to provide a proper response to the user via a Catch block.

Of course, having a WebHandler available to do work means nothing if we cannot provide security around the necessary actions. To that end, we can utilize features within PFMO to assert the current requesting user's identity and to re-instate any available context data for their current session. This is the topic of the next section.

Integrating PMFO

Integrating the PMFO stack only requires 2 things: the correct procedure library (PL) files in the PROPATH, and a Client-Principal Object. We will assume the former has been handled as part of Appendix A and other supporting chapters. To obtain a CP token, the most direct way within the ABL is to use the **OpenEdge.Security.Principal** class. This is a self-instantiating class which can take the session request object as input to a static Import() method. This produces a new Principal object that contains the CP as a Token property. The sample code below illustrates this process, as it would be utilized from within a WebHandler class.

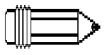
```
define variable oPrincipal as OpenEdge.Security.Principal no-undo.
oPrincipal = OpenEdge.Security.Principal:Import(session:current-request-info).
```

The following code asserts the Principal object's Token via the SessionManager's method called **establishRequestEnvironment()** which will assert the identity against any connected databases. If the identity is invalid or cannot be used with the available databases, an error will be thrown. On success, the PMFO stack will be available for use with all of its included managers. The code just above and just below should be included at the start of any Handle* methods of your WebHandler, ensuring that the request is properly secured and the requesting user is known to the application.

```
Ccs.Common.Application:SessionManager
       :establishRequestEnvironment(oPrincipal:Token).
```

At the end of each Handle* method there should be a FINALLY block which provides us a reversal of the above process. This should execute the **endRequestEnvironment()** method of the SessionManager, and delete the Principal object created previously (lest we have a lingering CP token in memory). This re-apply the "reset" token to the session and properly ends access for the previous user. It is also a good idea to clean up any other objects created as part of your response process for the WebHandler in this block.

```
Ccs.Common.Application:SessionManager:endRequestEnvironment().
delete object oPrincipal no-error.
```



NOTE

The PMFO stack is still available and "running" as of this point, we've merely ensured that the session information has been cleaned up after the request completed and a response was sent back through the server. You may still run any ABL code you wish after the above method is called, though you would no longer be associated with the running session for database access, etc.

Customizing the DataObjectHandler

In terms of WebHandlers, a special implementation exists as the DataObjectHandler (DOH) class and is utilized for the ABLWebApp project type to expose Data Objects via the WEB transport. This class is unable to be modified, as it exists within the OpenEdge core code. However, it does trigger certain class events which can allow for overriding of certain behavior. The following events in particular are highly useful:

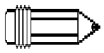
LoadEntity – When the resource (business entity) is loaded for a request.

Invoking – Before any method is run within the resource.

Invoked – After any method is run within the resource.

OperationError – After any error is raised from a resource method.

UnloadEntity – When the resource (business entity) is unloaded. (Added in 11.7.2)



NOTE

In this section, the term "invoke" does not refer to a type of JSDO-compatible methods, but of the general "dynamic invocation" of class methods. Therefore, the "invoking" and "invoked" events would be triggered on the create, read, update, delete, and submit methods as found in a business entity, and yes even the actual "invoke" types.

Following the previous process for securing a WebHandler and for making use of the PMFO stack, we can now do the same using the DOH class. This can be accomplished by 2 primary steps: create a simple ABL Class that contains methods subscribed to above events, and a Session Startup Procedure that creates this new class as part of each session's startup process.

Event Handler Class

The DOH event handler can be a simple ABL Class and does not need to inherit any other classes. However, because of the need to start it up with each session it must be made "immortal". This is simply done by providing a class variable that will hold a reference to the class itself. This can be assigned in the class constructor, which will cause a circular reference that prevents the garbage collection (GC) of the AVM from removing the class.

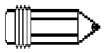
```
define private variable oCheat as DOHEventHandler no-undo.
constructor public DOHEventHandler():
    assign oCheat = this-object.
end constructor.
```

To subscribe to the events of the DOH, add the following to the class constructor as well. These address the respective class events of the DOH and subscribes a local method to each event.

```
OpenEdge.Web.DataObject.DataObjectHandler:LoadEntity
       :Subscribe(this-object:LoadEntityHandler).
OpenEdge.Web.DataObject.DataObjectHandler:Invoking
       :Subscribe(this-object:InvokingHandler).
OpenEdge.Web.DataObject.DataObjectHandler:Invoked
       :Subscribe(this-object:InvokedHandler).
OpenEdge.Web.DataObject.DataObjectHandler:OperationError
       :Subscribe(this-object:OperationErrorHandler).
OpenEdge.Web.DataObject.DataObjectHandler:UnloadEntity
       :Subscribe(this-object:UnloadEntityHandler).
```

Each method accepts 2 parameters: the object that sent the event and an event arguments object. Depending on the event, the latter may be handled by a slightly different class. Please refer to the following method signatures for each event type.

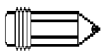
```
method private void LoadEntityHandler (
  input poSender    as Progress.Lang.Object,
  input poEventArgs as OpenEdge.Web.DataObject.HandlerLoadEntityEventArgs ):

method private void InvokingHandler (
  input poSender    as Progress.Lang.Object,
  input poEventArgs as OpenEdge.Web.DataObject.OperationInvocationEventArgs ):

method private void InvokedHandler (
  input poSender    as Progress.Lang.Object,
  input poEventArgs as OpenEdge.Web.DataObject.OperationInvocationEventArgs ):

method private void OperationErrorHandler (
  input poSender    as Progress.Lang.Object,
  input poEventArgs as OpenEdge.Web.DataObject.HandlerErrorEventArgs ):

method private void UnloadEntityHandler (
  input poSender    as Progress.Lang.Object,
  input poEventArgs as OpenEdge.Web.DataObject.HandlerLoadEntityEventArgs ):
```



NOTE

Because the UnloadEntity event was added in 11.7.2, it is not available in 11.7.0-11.7.1!

With the handler methods in place and events subscribed, we can now implement further changes. For our purposes, it is not necessary to perform any further changes within the LoadEntityHandler or the OperationErrorHandler methods—we wish to focus primarily on the **InvokingHandler** and **InvokedHandler** methods. Similar to the process with a simple WebHandler we need to obtain the user's CP token and assert the user's identity for security purposes. In other words, we can run the **establishRequestEnvironment()** method within the **Invoking** event handler, and the **endRequestEnvironment()** method within the **Invoked** event handler.



NOTE

In the past, the only way to obtain a CP token and assert identity was in the Activate and Deactivate events of the AppServer. This meant creating 2 distinct procedures that could not easily create the Principal object at the start of a request and delete the same object at the end of the request. By using the event handlers above we can create a private variable for the class and essentially "share" the Principal object between events (invoking/invoked), cleaning up after ourselves when complete. This works for us because only 1 request can be processed at a time by a session, so we're only holding the CP token only for a single user for a single request. And similar to the activate/deactivate events, use of the DOH events means this action takes place no matter what business entity is executed via the DOH.

Integrations with CCS

In the 11.7.2 service pack, the Common Component Specification (CCS) classes have been included within the OpenEdge environment. These classes are compiled and present in the OpenEdge.Core.pl file in a similar manner to the Ccs.pl that is commonly deployed with PMFO. As part of this inclusion, the DataObjectHandler and related OE classes were adapted to look for and utilize any available implementation of the Ccs.Common.Application and notably the ServiceManager class. If utilizing the full PMFO stack, there is nothing that needs to be configured further in the environment. However, if intending to use the OpenEdge version of the DataObjectHandler there are 2 critical items that must be considered.

1. Specify an implementation of the IServiceRegistry and IWebHandler classes.
2. Specify the lifecycle for the Service Registry implementation and its loader.

Both items can be handled within the service.json configuration file for the ServiceManager implementation. An example of these settings can be seen below, which is a perfect example of how this feature should be implemented. The ServiceMapping portion of the config tells the ServiceManager where to find the appropriate implementation of the given interface class, which should result in starting an instance of said class. The ServiceLifeCyle portion tells the ServiceManager how long to keep those instance—in this case the two listed classes should be kept for the entire life of the MSAS session. The default lifecycle (when none is specified) is only for the life of a request but this causes the ServiceRegistry to be destroyed in the process, as well as all information regarding the available endpoints for that service. To avoid this issue, the two stated service classes will remain running even when a request completes.

```
{
    "Config": {
        "ServiceMapping": [{
            "Service": "OpenEdge.Web.DataObject.IServiceRegistry",
            "Implementation": "OpenEdge.Web.DataObject.ServiceRegistryImpl"
        }, {
            "Service": "Progress.Web.IWebHandler",
            "Implementation": "OpenEdge.Web.DataObject.DataObjectHandler"
        }],
        "ServiceLifeCycle": [{
            "ServiceMatch": "OpenEdge.Web.DataObject.ServiceRegistryImpl",
            "LifeCycle": "Session"
        }, {
            "ServiceMatch": "OpenEdge.Web.DataObject.ServiceRegistryLoader",
            "LifeCycle": "Session"
        }]
    }
}
```

NOTES: