Appendix C

Progress Modernization Framework for OpenEdge

Catalogs and Metadata

Purpose

This addendum describes the process used by OpenEdge and PMFO to generate and expose metadata for RESTful endpoints. It should provide a greater understanding of how HTTP artifacts are mapped to ABL code, and the patterns utilized by the JSDO to understand and access those endpoints. Both the pre-generated JSON catalog from PDSOE and the runtime-generated catalog from PMFO will be compared and contrasted to illustrate some pros and cons of each approach, as encountered during various real-world implementations.

Disclaimer: Under no circumstances should a conclusion be drawn that one solution is better than another. Each will be proven to have their own strengths and flaws, while continuing to improve upon any weaknesses with every code iteration. Concerns found in 2017 may no longer be issues in 2018 or beyond, while new functionality may bring about interest in other areas for comparison.

RESTful Interfaces

Before we can expose a service, we need to understand RESTful requests and examine the components involved. In the following examples we'll assume that we are building something that nearly everybody has experience with: an interface to track a shipment. This will be a **public** API that accepts a **tracking number** and should return the **current status** of a matching shipment along with a **history** of where it has been. Since there is no true standard for REST, we can assemble any URL we wish. But for maintainability and consistency, we need a pattern that makes logical sense and can be adapted easily to other situations. In "traditional" patterns, the following could be a perfectly valid scenario:

```
GET http://www.mydomain.com/api/tracking/1Z2345
```

This gives us a distinct server (fully-qualified domain name, or FQDN), identifies that we want an API interface (not a webpage), requests a specific endpoint to "track" something, and a value for inquiry. Though this still leaves a critical question unanswered: what is it that we are tracking—was it an order ID or something else? This is where we have to start getting a bit more descriptive in our URL scheme, and adapting to a more common pattern seen in OpenEdge for REST endpoints.

```
GET http://www.mydomain.com/web/public/order/track?id=1Z2345
```

or

```
verb scheme://server/transport/service/resource[/operation][?params]
```

So, what is different here? We still have the same server, but we changed the first subdirectory from "api" to "web". Within the PAS technology, this is called the **transport** and can be either "rest", "web", "apsv", or "soap". For our immediate purposes **rest** and **web** can accomplish the same task, though we will be focusing on use of the "**web**" transport which is preferred. [At least as of 11.6.3, and for reasons which will be explained shortly.] The next item in the URL is our **service** named "**public**", which should indicate we are working within the realm of a public interface. This could have been called "orderservice" or "tracksvc" or any other appropriate value, but for our purpose will allow for the whitelisting of all resources under this URL to be open to the public. Speaking of which, next we have a **resource** called "**order**" which implies that we are about to perform some type of **operation** against an order—specifically, to "**track**" one. Lastly, to qualify which order we intend to track, the explicitly-named **parameter** "**id**" will provide that piece of data.

To turn this URL into more of a natural statement by an end-user:

"I want to **get** the **public** data for an **order**, with intent to **track** an order **ID** of **1Z235** via the **web**".

Or translated into terminology as viewed by PAS:

"When a GET verb is used against the **track** method in the **order** resource, under the **public** service, via the **web** transport, obtain the **ID** property as an input parameter and return any output parameters".

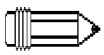
In the next sections, we'll investigate how metadata performs this translation between a URL endpoint and PAS (ABL Code), and how a catalog provides the user-friendly description of these endpoints.

Progress Data Object Service

The specification for a **Cloud Data Object (CDO)** (http://clouddataobject.github.io/) states that it "provides data management features normally found in Systems of Record to client applications in the cloud, specifically web and mobile apps". Within an OpenEdge environment there are 2 sides to this feature: server-side, called the **Progress Data Object (PDO)**; and client-side, called the **JavaScript Data Object (JSDO)**. Naturally, we will be focused on the PDO here, which provides a server-side resource with operations for managing data through a REST web service. This is accomplished by use of a user-defined ABL class called a **Business Entity**, and as of OpenEdge 11.6.3 there are 2 means of creating a valid Data Object: directly in PDSOE or through helper classes in PMFO.

Pattern:	**PDSOE**	**PMFO**
URL Base:	/web/pdo/<service>/<resource>	/web/pdo/<service>/<resource>
ABL Class:	OpenEdge.Web.DataObject.DataObjectHandler	Spark.Core.Service.DynamicEntity
Supports:	Read-Only, CRUD[+Submit], Invokes	Read-Only, CRUD[+Submit], Invokes

As illustrated above, both means of producing a web-based Data Object that can support the same critical functions: A Business Entity which may consist of a Read-Only ability, CRUD operations, a Submit operation, and any arbitrary Invoke operations. The exact means of how this is managed via metadata and any associated catalog will be covered in the next sections.



NOTE

Use of a PDO is by no means the only way to create a valid RESTful interface. Use of the **WebHandler** class in 11.6+ can parse and accept any URL via any HTTP Verb, but is not the focus for this section. This is the primary focus of **Appendix B** of this training material, so please refer to that chapter regarding the customization of WebHandlers, if you have not already.

REST vs. WEB Transports

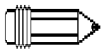
Within PDSOE there are 2 means of exposing a Data Object (Business Entity): via REST and WEB transports. This is worth mention as these transports have occasionally caused confusion since the introduction of the "web" transport in 11.6, initially for "WebSpeed for PASOE". However, both transports provide exposure of RESTful interfaces using a URL scheme that is 99% identical. Or stated another way, if the same Business Entity were exposed using both transports, access to the exposed operations would only differ by at most 2 elements in the URL, but should execute the associated ABL code in the same manner.

To illustrate the differences between the REST and WEB transports, let's compare and contrast some of the key elements for each, notably the way metadata is generated, stored, and utilized to execute ABL code. Some of these options are more compatible with a multi-developer project, in terms of how metadata can be accessed or overridden after initial generation by PDSOE. The preference as of 11.6.3 is the use of the WEB transport, due to its use of an ABL class to manage requests and the easily-overridden nature of the metadata.

Transport:	REST	WEB
URL Base:	/rest/<service>/<resource>[/<method>]	/web/pdo/<service>/<resource>[/<method>]
Metadata:	Generated by PDSOE	Generated by PDSOE
Storage:	Encapsulated in a .paar file in WebApp	.gen file from PDSOE, with .map override
Usage:	Maps URL to AppServer via Java client	Maps URL to DataObjectHandler class
Retention:	Cannot commit to source control	Can commit .gen/.map to source control
Updates:	Re-run the ABL Service wizard	Re-run the ABL Service wizard

Generating Metadata

Regardless of how a Data Object (Business Entity) is created, there must be a relationship between a URL pattern and a distinct class method or internal procedure in ABL. This is necessary as we are moving requests between two different systems: the web as handled by Tomcat, and the ABL as handled by an AppServer (the Multi-Session AppServer). In order to accomplish this bridging of systems, metadata must be created to describe the available mappings.



NOTE

It is absolutely critical to separate the idea of metadata from that of a catalog:

Metadata is a collection of internal mappings used to identify the code to be executed if given a specific URL.

A **catalog** is a public representation of all available URL patterns which will be understood and mapped by the metadata.

The latter is only used by end-users or client-side code to know what services are available. In other words, a catalog is merely a convenience to the front-end and may be completely omitted, but metadata must **always** be present in order to have an operational endpoint.

To illustrate the differences between metadata generation and usage between traditional PDSOE (11.6.3+) and the PMFO patterns, please consult the table below. Both development options produce a compatible means of mapping HTTP artifacts to ABL class methods, and to execute class methods (by default) by use of dynamic invocation. **No matter how the metadata is generated, stored, or utilized, the process for executing ABL code by use of each approach is essentially the same.**

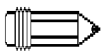
Pattern:	PDSOE	PMFO
Classes:	DataObjectHandler	CatalogManager.cls + Reflection.cls
Tooling:	Built-in (Create/Edit an ABL Service)	Built-in (Within the PMFO.pl code)
Generated:	At development, pre-deployment	At runtime, post-deployment
Format:	JSON file (.gen with .map override)	JSON data (while processing)
Access:	Read by DOH class at first run (or startup)	Read by RouteManager.cls, on-demand
Caching:	Datasets in MSAS session memory	Datasets in MSAS session memory
Usage:	Must parse URL, locate mapping, create class instance, execute ABL	Must parse URL, locate mapping, create class instance, execute ABL
Execution:	Invoke class dynamically, passing params	Invoke class dynamically, passing params
Response:	Gather output params and return	Gather output params and return

Behind the scenes, much of the technology is the same. Mapping data is somehow read or generated from a Business Entity to be exposed in a service. This data is read (typically at session startup), and then processed into internal datasets for easy access at runtime. From that point forward, every URL processed through a request will identify a mapping to an ABL class, gather the necessary parameters to be passed (from the HTTP request), dynamically create a new instance of the target class, and perform an invoke on the necessary method. Though beyond the metadata being generated, there are some subjective pros and cons to each approach for creating and storing this information.

Pattern:	PDSOE	PMFO
Creation:	Must be re-generated after changes to method signatures, re-published	Read at runtime after code is re-published (and sessions trimmed)
Source Control:	Yes, but merge conflicts are possible with multiple developers	Not possible, but also not necessary since all is generated at runtime
Application Stack:	All features come standard within OpenEdge release within PDS/ABL	Features must be added via PL files to PROPATH, configuration files
Bugfixes/Enhancements:	Available only with new OpenEdge releases or service packs	Available as hotfixes by customer need, or part of regular releases
Extensibility:	DOH class untouchable, but can utilize class events to override	All code can be overridden directly, or inherited and extended

The Data Object Service Catalog

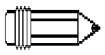
Catalogs for API's may exist in any number of formats. Within the OpenEdge environment this format is the **Data Object Service Catalog**, which reflects the service-oriented approach of the Cloud Data Object. For other systems, a more open API format such as Swagger, WADL, or RAML may be used, especially by those who employ an "API-first" approach. Regardless of format, the purpose of a catalog is to provide documentation about your back-end services.


NOTE

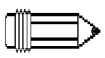
Within PDSOE, the process for generation of a catalog pre-dates the DataObjectHandler class its generation of the mapping files (metadata) behind that feature. Therefore, these are currently and foreseeably separate processes and do not depend on each other. When an ABL Service is created or updated (for purposes of use with the "web" transport), both processes will be run against the selected Business Entities to generate the respective content. This separation is due to the need to only satisfy the default behavior of the PDO and JSDO per the Cloud Data Object specification. In the event that a different catalog format is needed, either PDS would need to be updated to provide that option or the mapping data would need to be processed to generate the necessary output.

As with the metadata, there are some pros and cons to each approach for the generation and formatting of the catalog data. Granted, these may be subjective depending on your needs, but are interpreted here in terms of maintainability or accessibility of each aspect. Of course, there are some aspects which are neither a strength nor weakness and will be shown in blue.

Pattern:	PDSOE	PMFO
Generation:	Only from within PDSOE	At runtime, external to PDSOE
Compatibility:	Guaranteed to work as part of QA process for OpenEdge releases	Must be updated when specs are available, possibly post-release
Format:	Only PDO/JSDO	By default, PDO/JSDO
Extensibility:	None, can only output the single syntax/format for PDO/JSDO usage	CatalogManager can be replaced with any custom implementation necessary
Output:	Static JSON file in /static/ directory	JSON output via web request
Performance:	Static file can be cached by browser	Re-generated on each web request
JSDO Behavior:	Cannot alter flags/options in catalog unless a related annotation exists	Can freely modify output as necessary to drive JSDO features (eg. req/resp wrap)


NOTE

As of the time of this material's creation, not every possible annotation available has been documented, or at least not in context of a real-world use-case. Most crucial items are listed in the latest documentation for OpenEdge 11.7, but not all within the same section of the documentation. It may be necessary to hunt for some items between the formal documentation portal and the Progress Communities site.


NOTE

As of OpenEdge 11.7 there is an ANT task available to generate catalog data from anywhere via the command line. However, this does not address the generation of the initial "<service>.gen" files as would also be created from the standard BusinessEntity annotations. Eventually this may be supported, but at this time is not known to be on the roadmap.

Extending the Catalog

At present, there are no available examples of extension for catalog generation as it is performed by PDSOE. However, in terms of the PMFO the process to create a customized catalog would involve overriding the **CatalogManager** class. Beginning with the **startup.conf** file, a developer would modify the JSON data to specify a custom implementation of ICatalogManager as shown below:

```
{
    "Manager": "Spark.Core.Manager.ICatalogManager",
    "Implementation": "Custom.CatalogManager"
}
```

With this new *Custom.CatalogManager* class in place, it would then be possible to inherit the original **Spark.Core.Manager.CatalogManager** class and simply override the **getCatalog()** method. This is the top-level method that produces a JSON object with catalog data, and is called automatically within PMFO when using that version of a dynamic service interface (eg. the web/pdo/ URI). How the JSON data is constructed is up to the developer—it could be a standard format such as OpenAPI or Swagger, or any custom output in JSON format.

The source of data for this new output would come from the metadata stored in the internal temp-tables ServiceList, ResourceList, ServiceInfo, ResourceInfo, MethodInfo, ResourceProperty, and EntityProperty. In addition, classes such as the SchemaManager could be called upon to return detailed information about dataset/temp-table structures that map to certain method parameters. Examples of how these internal temp-tables are used can be found in the private helper methods getServices(), getResources(), getSchema(), getRelations(), getOperations(), and getDataDefs(). While these are meant to return data in the format expected for PDO/JSDO usage, the process of building up the expected JSON object/array data can be gleamed from the original source code for the class.

NOTES: