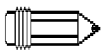
Chapter 4

Progress Modernization Framework for OpenEdge

The PMFO Framework

Purpose

Thanks to our wizard, we have already started utilizing the PMFO architecture and supporting ABL classes for CRUD usage. What we have not discussed is the "why" and "how" behind all of this. This chapter will further explain what classes area already in use, including our various manager classes, session startup/shutdown logic, and even some advanced security topics (read: the **OERealm** security model).



NOTE

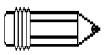
Before we proceed, it is assumed that the necessary code is already present and ready to use. However, in the event you did not start with the earlier chapters or you wish to include the PMFO ABL libraries in another project, it is necessary to first follow the section "**Obtaining Source Code**" in **Appendix A**.

Including the PMFO Framework

When we created the initial training project we tailored the directories and included a directory "**PASOEContent/WEB-INF/openedge/Business**" under the initial structure. This was in anticipation of being able to include additional libraries without interfering with our existing application code. These steps would be necessary for any new project, where you wish to start from scratch**—we already performed these steps for TraingSample by copying from an existing project!**

1. Using the checked-out code from the repository, we will navigate to the **C:/Modernization/spark/framework/server/** directory.
2. Copy the files **Ccs.pl** and **PMFO.pl** to your "**ABL Web App**" project's **/AppServer/** directory, and include these (in order) at the end of your project's PROPATH.
    a. Note that these should be manually deployed to the **CATALINA_BASE/openedge** folder of your PAS instance as they are shared libraries for all WebApps.
3. Copy the contents of the **samples/Conf/** directory to your project's **Deploy/Conf/** folder. This will provide the necessary default configurations for the manager classes.
4. And finally, we need to improve the functionality of our AppServer's security, so we will utilize some session procedures under the new **Spark** directory to accomplish this:
    a. The **Spark/startup.p** will look for any startup params (in JSON format) that affect operation of the MSAS Agent's sessions. It also starts the SessionManager and loads any business logic into the CatalogManager.
    b. The **Spark/shutdown.p** will attempt to shut down any remaining services or managers in memory, and if logging is set high enough will also dump information about any objects still in memory (a very useful tool for tracing memory leaks in your application).

That wraps up the basic inclusion process. Now we can focus on modifications to the configuration of the managers, and complete the setup of our new security and session/context layer.



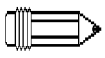
NOTE

All files within the PMFO ABL libraries are open and unencrypted for viewing. That means you can see exactly what each class should be doing, and freely modify this code. However, for purposes of release management and avoiding conflicts with your custom code, it is highly advisable that you instead create your own implementations of any classes that require customized code. This process will be fully explained in the next few sections.

Introducing the Managers

Everything within the PMFO ABL code starts with a manager class, and nearly everything is configurable and can be overridden. But before we can configure, let's introduce the classes:

- **StartupManager** – This is the primary class that drives all other managers. It's a class accessed via its own static **Instance** property, which creates the class instance if it does not already exist. It is during this instantiation process that various configuration files are read from disk and can alter the behavior of the class and its descendants.
- **RouteManager** – Called by the service interface, and determines how to execute the request. Typically creates the necessary service implementation via the ServiceManager, first.
- **ServiceManager** – Creates a service implementation and can execute said service for a given lifecycle. This is an open-ended and generic class.
- **CatalogManager** – Provides a means of automatically registering ABL classes or procedures as REST resources, and producing a catalog structure as required by the Progress JSDO.
- **ConnectionManager** – Creates an external connection to any service as defined in the configuration. A typical use is for making additional AppServer connections.
- **LoggingManager** – Fairly self-explanatory, provides logging capabilities within the framework. It is also used to capture errors and handle certain types in a specific manner.
- **MessageManager** – In PMFO terms a message is a particular instance of a request or response object. This manager provides handlers for certain built-in types that handle more complex operations than the standard JSON request/response.
- **SchemaManager** – Utilized by the **CatalogManager** to dynamically register schema information either from a connected database or an included dataset or temp-table definition.
- **SessionManager** – Creates and manages user context within the application, after asserting the identity of the user against the connected database(s). Also provides methods to set or reset context attributes.
- **StateManager** – Utilized by the session manager to read/write session data. When a session is started the **SessionManager** creates the initial object, and this class stores the data (default: flat file) when the session is ended. When re-establishing a session, the **SessionManager** reads the existing context and populates the context object. This class would typically be overridden, for example, to provide that context storage within a database.
- **TranslationManager** – Provides a single point of override for translating text.
- **StatsManager** – Tracks common request/response information for reporting and statistics purposes, though by default will provide some level of debugging when the agent logging-level is set to 3 or higher.



NOTE

Most managers are driven by a configuration file that should be named accordingly with the class it represents. For instance, SessionManager has a session.json config, StartupManager has a startup.json, etc. These may be found in the **Deploy/Conf** folder of the project, and must be copied to your PAS instance's **conf/spark/** folder (unless this is overridden with an alternate location).

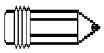
Security Best Practices

If continuing with the original training project **TrainingSample**, we should have already configured the AppServer to utilize the Activate/Deactivate procedures. Likewise, the connected database should be configured with at least one EXTSSO (_extsso) domain called "*spark*", which was done as part of the setup for the original WebWizardAdv project demo. However, in the event this has not yet been performed or you are starting a new project on another environment, please complete the instructions for "**Domain Configuration**" under the section "**WebWizardPAS Setup**" in **Appendix A**. For security best-practices this is a highly recommended starting point to create a domain, as we will require it next.

Client-Principal Tokens

To recap from earlier in the material, when a user authenticates through Spring Security in Tomcat a special identity token is created and then sent with the request to the application's AppServer. This may be referred to as a session token, and is obtained as an ABL Client-Principal Object in the AppServer's session request handle (**session:current-request-info:GetClientPrincipal()).** The other typical form of a CP token is a static file that can be generated to disk. Within the **<PAS>/conf/spark/** directory (which should be your go-to configuration location) are some TXT files that explain how certain CP tokens will operate. In our application we will be creating two necessary static files: one that represents an **unprivileged** user, and another that represents an **anonymous** account. This will allow us to present a total of three different user types to the application:

1. A known, "*authenticated*" user will have their session CP token as identification.
2. An anonymous user will be from a "*public*" request (a known-unknown user).
3. A "*reset*" token will return an AppServer agent to an unprivileged state.



NOTE

There is a known issue with the **GetClientPrincipal()** method that affects some older versions of OpenEdge, and may still be present in 11.7. Each request to this method creates a new instance of a Client-Principal object which remains in memory until deleted. Therefore, it is advisable to use this method as few times as needed, and instead pass the resulting CP object between programs that require it. Always delete your CP objects when completely finished with their use.

Per the instructions in the readme files, we can create CP tokens for the PMFO configuration files.

Note: You only need to perform these steps if the necessary files are not already present!

1. From the **C:/Modernizaton/TrainingSample/Deploy/Conf** directory, **Shift + Right-click** on the **Realm** folder and select the option "*Open command window here*".
2. Run the following command to create our general realm token for authentication use ONLY:

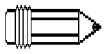
   ```
   genspacp -password sparkRealm -user sparkREST -file SparkRealm.cp
   ```

   a. The password used here should be **different** from any other in use in your application. Note that this token does not even include a domain!

   b. Copy the generated "SparkRealm.cp" file from **Deploy/Conf/Realm** to the folder **CATALINA_BASE/common/lib/** so that it is available to Tomcat.

3. Next run the following command to create the reset token, which will remain in the **Conf** directory as this will only be used by the AppServer (not Tomcat like "SparkRealm.cp"):

   ```
   genspacp -password spark01 -user sparkREST -role NoAccess
            -domain spark -file SparkReset.cp
   ```



NOTE

It is important to remember that the Spring Security framework **does not** actually perform any authentication actions itself. It is merely a conduit to the logic that *does* perform these actions. What we are preparing here is the security over the conduit itself, the class that will perform our authorization of user credentials, and the information to apply to the returned token to make our user compatible with the database (domain) to be used by our application.

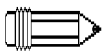
Applying the Configuration

With the tokens generated, we would put them into play via several PMFO configuration files:

Note: We already have valid files available for our purposes, so this part is for reference only!

1. In the **Deploy/Conf/** directory, open the file "*sesison.json*" and set the "*ResetClientPrincipal*" value to "*SparkReset.cp*". This provides the name of a static CP token that will reset the security on the AppServer session when each request ends. Save any changes made.
2. Next, the file "*SparkRealm.json*" requires that you supply the encoded (oech1) password for "*password*" that came from our generation of *SparkRealm.cp*. Save any changes made.
3. In the file "*session.json*" be sure to add a new object in the "*Domains*" array that specifies the "*domain*" as "*spark*" and the "*accessCode*" as your encoded (oech1) password. Save the file.
4. In PDSOE, open the "**PASOEContent/WEB-INF/oeablSecurity.properties**":
    a. For the "**OEClientPrincipalFilter.domain**" property, set the value to "spark"
    b. For the "**OEClientPrincipalFilter.key**" property, set the value to your encoded passcode (the ohch1 value).
    c. Save the file.
5. With the files modified, copy the .JSON and .CP files from your project **Deploy/Conf/** directory to your PAS instance under the **conf/spark/** directory.
6. For best results, we should make sure we fully pick up the changes, so we should restart our PAS instance. This can be done via the **Servers** view in PDSOE.

With the server restarted, we can confirm if our application is working as expected. To do this, return to the website at **http://localhost:8830/TrainingSample/static/login.html** and authenticate using the "*dev/bravepoint*" account, and allow the application to load. We can then look at the file **C:/PASOE/TrainPAS/logs/TrainPAS.agent.log** and confirm that the framework logic has output information about the Client Principal Object. If working correctly, we should see the username, domain, and a session ID output in the file. If there were any issues with the domain or key provided, then there would be errors about validation of the CP token. If no issues are present, then we have successfully configured the AppServer and security for this application. Now we can move on to customize our manager logic, knowing that we can accurately establish the identity of the user.



NOTE

This example covers the applying of local security, which simply uses a flat file for storing credentials. If we were instead using **OERealm** security (for example) we would need to access another external AppServer to validate credentials. Again, this is because Spring Security does not actually perform the authentication—that would be the job of the remote service in this case. And that means the remote service cannot just be wide open for anyone to request data, or else it would be abused by denial-of-service or brute-force attacks. Providing a CP token between Tomcat and the remote server insures that requests only come from a known source, and that both parties are using the same shared key for identification. And use of the CP token on disk and the passcode phrase in the configuration files means we don't have to store any passwords in the clear.

Current OO Patterns

At this stage the application can utilize any of the features of the PMFO ABL libraries, though it's worth noting exactly how to incorporate them into your existing **Business Entities**. What we have set up so far is a means of modifying our session and context at the start of an agent request, allowing for common overrides to be applied if needed before any business logic is run. It also means that all the manager instances are ready to be utilized, and the user's identity already asserted against the connected databases (via the *set-db-client* function). It is important to note that as of version 3.0 of the PMFO, we utilize the **Ccs.Common.Application** class as the primary starting point for finding our way. This enables you to make use of any manager from anywhere in your code by first accessing the necessary static property of this class.

Understanding the CCS Classes

During development of PMFO, work in the OpenEdge community was being done against the Common Component Specifications (or CCS for short). This open effort was to designate a common set of class interfaces that would make it easier for everybody to create their own custom implementations of any manager class. To that end, PMFO now incorporates (read: depends) on the CCS interface classes which are included in the Ccs.pl file seen throughout several of our configuration steps. This library contains no real executable code, though static properties are certainly used in the primary class, **Ccs.Common.Application**, which contains 3 critical managers:

- StartupManager as Ccs.Common.IStartupManager
- SessionManager as Ccs.Common.ISessionManager
- ServiceManager as Ccs.Common.IServiceManager

This is initialized by a single line of code, typically in the Session Startup Procedure, **Spark/startup.p**:

```
Ccs.Common.Application:StartupManager =
Spark.Core.Manager.StartupManager:Instance.
```

The property Instance on the PMFO **StartupManager** is self-instantiating, which returns an instance of the manager class. This in turn starts all other managers and either holds their instance references internally in **StartupManager**, or in the case of the **SessionManager** and **ServiceManager** classes, assigns their instance directly back to the properties in **Ccs.Common.Application**.

Example: Using the ClientContext Class

As an example, let's say you want to obtain the name of the current user. This is stored within the user context, which is accessed as an instance through the static **CurrentClientContext** property, which is held by the **SessionManager** class instance. To gain access to this, we need to construct a route down to that property via our managers:

```
using Spark.Core.Manager.IClientContext from propath.

…

define variable oClientContext as IClientContext no-undo.

oClientContext =
cast(Ccs.Common.Application:SessionManager:CurrentClientContext,
IClientContext).
```

Here our first step should be to simplify the code by adding a USING phrase at the top of our file. Next, we need to access the instance of the **SessionManager**, which contains the instance property **CurrentClientContext**, which is defined as type **Ccs.Common.IClientContext**. Because this class and its interface can be extended, we need to cast this instance against the expected PMFO interface. With this property cast and assigned into a variable that is strongly-typed as our expected **ClientContext** class instance, we can easily access any property it contains. Luckily through PDSOE we have a lot of help via code hinting, so by just pressing the ":" key after an object instance we can get a list of properties and methods available along the way. In this case, we would want to access the **userID** property, which is always up to date with the current username from the CP token. By comparison if we wanted to obtain a custom value we could instead use the "*getUserProperty()*" method on the context object to return user-defined properties.

Example: Using Other Manager Classes

To access other managers, we must follow a similar approach. For the **SessionManager** and **ServiceManager** classes, these are very simple. These exist as static properties directly found in the **Ccs.Common.Application** class, though they must still be cast to their expected PMFO interface:

```
cast(Ccs.Common.Application:<mgr_property>, <pmfo_interface>)
```

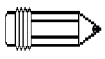
This pattern is similar for most of the remaining manager classes, though these will always be accessed through the **StartupManager** property, and in turn through its **getManager()** method.

```
cast(Ccs.Common.Application:StartupManager
        :getManager(get-class(<pmfo_interface>)), <pmfo_interface>)
```

The method **getManager()** should return the implementation of a given interface class, pulling from the *StartupManager*'s internal registry of class instances. But because this method returns a generic *Progress.Lang.Class* datatype, it **MUST** be cast to the expected interface after the fact.

Introducing the Entity Classes

The first place we can make use of inheritance is by adopting the **Dynamic Business Entity** class, **DynamicEntity**. In our current project which utilizes the wizard, our business entities all inherit from this class to provide default behavior for Create, Read, Update, Delete, and Submit methods. This provides all the common methods that are expected by the JSDO within our application, and can be shared by all of our entities. This common class itself inherits from a **SparkEntity** class to provide even more generic helper methods for our data operations, such as a smarter "*filterData()*" method which can accept filter options either as Kendo criteria objects or pre-built ABL phrases. This class in turn inherits the standard Progress **OpenEdge.BusinessLogic.BusinessEntity** class and provides most of the automated CRUD+Submit operations on a dataset.



NOTE

While the PMFO class **DynamicEntity** provides default methods for CRUD and Submit actions, for situations where these methods are not required there exists the **DynamicResource** class. The purpose of this class is to provide an entirely custom starting point for exposed API's. All methods added to this class are expected to be Invoke type operations. It is still possible to add your own CRUD methods, these could utilize any input/output parameters you wish—by comparison those methods of DynamicEntity are strictly meant to work with the corresponding JSDO operations.

Overriding Core Classes

The next level of inheritance is actually to provide overrides to the existing manager classes. To start simply, we will override the **ClientContext** class to provide customizations for our application:

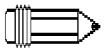
1. Right-click on the **WEB-INF/openedge/Common** directory and select **New → ABL Class**
    a. Provide a name of "**ClientContext**", and click **Browse** next to the "Inherits" field.
    b. Type "*ClientContext*" in the filter and select the original **Spark.Core.Manager.ClientContext** class. Click on **OK**.
    c. Click on **Finish** to create the class.
2. The file should be created, but it will likely have a compile error. We need to add some compatible constructors to support the inherited class, along with debugs to prove this works:

```
constructor public ClientContext ( ):

      message "In constructor".

end constructor.
```

    a. Save the file and confirm that the compilation errors are gone.
3. Open the file **Deploy/Conf/service.conf** to modify our new implementation. We utilize the service config file since the ClientContext is a service of the SessionManager class.
    a. For the **ServiceMapping** array, we need to add a new JSON object as follows:

```
{
      "Service": "Spark.Core.Manager.IClientContext",
      "Implementation": "Common.ClientContext"
}
```

    b. This will tell the framework that we are providing a new implementation of a service, which adheres to the interface as defined by IClientContext.
    c. Save the configuration file, and copy to the **conf/spark** folder in **TrainPAS**.
    d. Publish your class to trim all sessions on the **TrainPAS** server.
4. Refresh your previous browser tab with the application, and log in again if necessary.
5. Open the **TrainPAS.agent.log** file (if not still open) and confirm that the messages from the constructor appear. If so, then we have successfully created a custom override for the ClientContext class, and can now override any methods as necessary (eg. The method "*initializeUserProperties()*" which can set default values for the user when instantiated).



NOTE

It is not necessary to inherit directly from the overridden class, though not doing so implies that you intend to fully override the class and will need to copy the original class file from the Spark directory into your own local application folder. The approach outlined here is more flexible as it allows you to adopt and adapt any new methods or logic that appear in future revisions of the PMFO ABL libraries.

Upgrading to OERealm Security

At some point you will need to apply a more flexible form of security, most likely to utilize an existing table or authentication scheme. This would be the job of the **OERealm** security model and it combines many of the features illustrated so far. More importantly, this is closer to a "*Single Sign-On*" (SSO) pattern or authentication as you can run this class in its own AppServer and have it serve as the SSO for multiple other services. Though for simplicity we will run this within the same AppServer as our **TrainingSample** application. First we will set up the necessary code, then discuss how to modify the new security class to alter its default behavior.

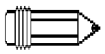
1. Within PDSOE create a new subdirectory under **WEB-INF/openedge** called "*Security*".
2. Copy the file "*SampleRealm.cls*" from the **/spark/framework/server/samples/** directory to your new **openedge/Security** directory as "**TrainingRealm.cls**".
    a. Open the file and adjust the class path of the file to be "*Security.TrainingRealm*"
3. Open the **PASOEContent/WEB-INF/oeablSecurity.properties** file (via double-click) and change the property "`http.all.authmanager`" to be "**oerealm**".
4. In the same file, look for the properties that begin with "**OERealm.AuthProvider**" and confirm that the following items are configured as shown.

```
OERealm.AuthProvider.createCPAuthn=true
OERealm.AuthProvider.sealClientPrincipal=true
OERealm.AuthProvider.multiTenant=false
OERealm.AuthProvider.userDomain=spark
OERealm.AuthProvider.key=oech1::23222e35397562
OERealm.AuthProvider.expires=0
OERealm.AuthProvider.authz=true
```

    a. Note: The properties for "**userDomain**" and "**key**" should always reflect the chosen domain name and encoded passcode for your application.
    b. Note: If using a multi-tenant database, the option for "**multiTenant**" would be set to "true" and a property "`OERealm.AuthProvider.registryFile`" added.
5. In the same file, look for the properties that begin with "`OERealm.UserDetails`" and confirm that the following items are configured as shown.

```
OERealm.UserDetails.realmURL=internal://nxgas
OERealm.UserDetails.realmClass=Security.TrainingRealm
OERealm.UserDetails.grantedAuthorities=ROLE_NoAccess
OERealm.UserDetails.appendRealmError=false
OERealm.UserDetails.propertiesAttrName=
OERealm.UserDetails.userIdAttrName=
OERealm.UserDetails.realmTokenFile=SparkRealm.cp
OERealm.UserDetails.rolePrefix=ROLE_
OERealm.UserDetails.roleAttrName=ATTR_ROLES
OERealm.UserDetails.enabledAttrName=ATTR_ENABLED
OERealm.UserDetails.lockedAttrName=ATTR_LOCKED
OERealm.UserDetails.expiredAttrName=ATTR_EXPIRED
OERealm.UserDetails.realmPwdAlg=0
OERealm.UserDetails.certLocation=
```

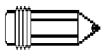
6. **Publish** the recent changes to *TrainPAS* via the **Servers** view, and **Restart** the server.
7. Clear your browser cache (to remove any old session info), and navigate to **http://localhost:8830/TrainingSample/static/auth/login.html**


NOTE

A major change between OpenEdge 11.6 and 11.7 is the consolidation of security options into these oeablSecurity CSV and Properties files. Notice that we did not need to modify some areas such as the "*OEClientPrincipalFilter*" properties nor the URL patterns in the CSV file. The new pattern for managing security should now involve only modifying the necessary items that correspond to your security model and authentication service, while security rules should remain intact and untouched.

If we now tried to access the application, we would be able to authenticate but we still won't proceed very far with requesting data from our API's. Why? This is because the default behavior of our **OEUserRealm** class is to blindly allow users to be authenticated, but does not grant any valid **ROLE** to the identity. What we need to do is modify the **TrainingRealm** class we created, and override some methods that are **OEUserRealm** class that we inherit from initially.

Similar to how a development instance of PAS is configured, everything in the Spark-provided **OEUserRealm** class runs in a wide-open mode. That means any value you pass to it will be approved. Send in any username, it will allow it. Send any password, it will allow it. Don't have a flag that represents whether the user is active? It'll allow it. You get the idea. So that's where we're at now: we can "authenticate" because the credentials are accepted blindly, but our user has no real privileges.


NOTE

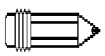
The inherited **Spark.Core.Security.OEUserRealm** class is a custom class in the Spark framework that implements the standard **Progress.Security.Realm.IHybridRealm** interface along with a custom **Spark.Core.Security.IAuthHandler** interface. The former identifies the signatures of the methods that match the requests from the Spring Security framework asking for pieces of data such as confirming the username, validation of a password, and returning various attributes. The latter is a set of helper methods that break down some of the tasks in *IHybridRealm* into simpler tasks.

So, let's update our custom OERealm class to return a known role for our user:

1. Open the "**TrainingRealm.cls**" file and add the following method:

```
method public override character getUserRoles
                        ( input piUserID as integer ):
    return "PSCUser".
end method. /* getUserRoles */
```

2. Save the file and republish to your PAS instance.
3. Return to the login page for your application, and log in again (forcing a new session).
    a. Confirm that you can now view the menu and access screens without issue.


NOTE

This is an example of overriding a class, you are implementing your own logic for the purpose of authentication and authorization. In this case we're returning a value "*PSCUser*" which will be prefixed with a value of "*ROLE_*" when assembling the roles for a user. The resulting "*ROLE_PSCUser*" role matches what is set in the intercept rule that applies to the generic **intercept-url** rule for the pattern **"/web/**"** which allows access if the user **hasAnyRole('ROLE_PSCUser')**.

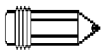
Understanding Intercept Rules

It is very important to understand how these rules operate, as they apply to ALL security models and can be a make-or-break moment in securing your application. Using the latest pattern from the oeablSecurity.csv file, they typically contain 3 attributes: **pattern**, **verb**, and **access**. This matches a URL pattern with a specific (or all) HTTP verbs, and allows access to the stated roles or condition. In OpenEdge versions prior to 11.7 these were handled via entries in each oeablSecurity-<model>.xml file and would have to be copied between files if the security model changed.

For URL patterns, you can specify a very specific path such as **"/web/pdo/common/customer**" which will match only the exact location. Something like **"/web/pdo/common/customer/balance**" would not be matched in this case and would fall through to a more generic rule. However, if you wish to match the parent URL only 1 level deeper you would use the pattern "**/web/pdo/common/customer/***" with a single "*". To match any depth level, you could use "**/web/pdo/common/customer/****" which would match a more complex endpoint URL such as "**/web/pdo/common/customer/balance/lastmonth**".

For the access property, the typical options are "*permitAll()*" and "*hasAnyRole()*". The first should be self-explanatory and allow access to the matched pattern by any role. The other option takes a list of separate roles, in the format of ('ROLE_1', 'ROLE_2', … 'ROLE_n'). Of particular note is that there is a very special role called "ROLE_ANONYMOUS" which must be in ALL-CAPS and represents an unauthenticated user. This is automatically applied to any user token (identity) that has not yet been authenticated. The benefit of this ability is that you can combine this role with a known role to allow access to an endpoint for both public and private users as illustrated here as found in the oeablSecurity.csv file:

```
"/web/pdo/*","*","hasAnyRole('ROLE_ANONYMOUS', 'ROLE_PSCUser')"
```



NOTE

Sequence matters for intercept-url patterns, as they will be processed in the order in which they appear. Therefore, the most specific patterns should come first and more restrictive/loose rules should come last. This is why the rules we added came before "**/web/****" so that any uncaught patterns will still be handled—in this case will require the role of "*ROLE_PSCUser*".

Further Realm Configuration

So what else can we customize in our **TrainingRealm** class? Well what about our user account? So far we're allowing anybody in, but we should probably only allow certain usernames. To do that, we need to override the **getUserID** method, which accepts a username and domain name as parameters. But what's important here is that we don't return a true or false response if the user is found, but instead we must return a unique numeric ID to represent the located user. This is because all other methods inside our class will be given this *UserID* value, look up the user record again, and then return a value. To illustrate, we can add the following method to allow only a specific username through as valid:

```
method public override integer getUserID
                              ( input pcUsername   as character,
                                input pcDomainName as character ):
    if pcUsername eq "dev" then return 1.
    return -1. /* Negative value indicates rejection. */
end method. /* getUserID */
```

With the new method in place, you should be able to trim your PAS sessions (via republish or restarting) and then log out of the application. Attempt a login again, though this time use something like "*foo*" as your username and confirm that the login is rejected. If this works as expected, then try the username of "*wizard*" again and verify that you can again log in—note that any password will work, as we have not overridden the method that validates that portion of the credentials. To better illustrate this, see the workflow below for each piece of data that would be requested from the Spring Security framework. For each attribute, a particular method will be called on the ABL side (defined by *IHybridRealm*), and each of those methods may call a more specific method in PMFO (defined by *IAuthHandler*).

Overall View:

[Tomcat → OpenEdge Default → PMFO Extension]

[Spring Security Element → IHybridRealm Method → IAuthHandler Method]

Specific Uses:

Username (w/ Domain) → ValidateUser → getUserID

ATTR_ENABLED → GetAttribute → isActive

ATTR_LOCKED → GetAttribute → isLocked

ATTR_LOCKED → GetAttribute → isExpired

ATTR_ROLES → GetAttribute → getUserRoles

Password → ValidatePassword → checkPasswordHash

If you look at the methods provided by the **OEUserRealm** class, you can override any that need to be for your application. Though most importantly these don't have to be driven by static values as we have illustrated here. You could use an attached database, or even read from a flat file—it's up to you and what is necessary for your application. In fact, you could even connect to another external service to perform the work—the only important item is the value returned by each method outlined in this class. The interfaces insure that the necessary methods are defined, and the default behavior insures that the scheme will work if not overridden with more specific code.

Advanced Topics

Creating API Resources

At this point we've discussed the **DynamicEntity** class for Business Entities in PMFO, and briefly examined the **DynamicResource** for Invoke-only purposes, but how can we utilize the latter to create fully functional API's? Luckily, a DynamicResource mostly acts like any other ABL class, where any public method will be discovered and exposed as an API endpoint as part of the framework startup process. It supports any primitive datatype (such as character, integer, logical, etc.) or complex data structure (mainly datasets and temp-tables). Some advanced classes are directly supported (like JsonObject and JsonArray), while any other datatype must first be converted to a string (as is the case with raw or handle variables).

For instance, a method created with a single input parameter will expect a JSON request with a property by the same name (case-sensitive) and datatype as the ABL parameter. Likewise, for a single output parameter from the same method, a JSON property should exist by the same name (and case) and appear in a format as allowed by JSON standards (eg. An ABL date value would be output as a string, in ISO8601 format).

Overriding PMFO Managers

Eventually there will be a need to adapt or alter the default behavior of the standard PMFO managers to suite your application needs. For example, the default behavior of PMFO is to track contextual information for a user's session by writing data to a flat file on disk. This is a perfectly usable solution and does not depend on any additional features for support. However, it can also lead to uncontrolled usage of disk space as well as not being very organized when hundreds of thousands of sessions may exist in a single day's requests.

A common example of overriding a manager can be found in the DynSports demo, where the **StateManager** class is overridden with a customized version. This utilizes the same methods for serialization/deserialization of context data, along with the storage and retrieval methods, but instead place this data into a table in the WebState database. In this way the same data can be more effectively and efficiently managed, while still utilizing the framework classes and remaining adaptive to potential fixes and enhancements from future releases.

Customized WebHandlers

Occasionally the need may arise to have a wholly customized interface for your business logic, using a pattern that does not fit with the CRUD/Submit/Invoke pattern. Thankfully the WebHandler class found in OpenEdge 11.6 and later allows you to create a nearly limitless means of dealing with requests over HTTP. This means getting and setting any headers or cookies, as well as processing and responding with any message body. And in conjunction with PMFO there are means of providing security through standard identity management, ensuring that any request to the server can be confirmed as belonging to a known user. Additionally, this allows your requests to be compatible with the remainder of the PMFO and its abilities, post-authentication.

In particular, what we need to do is assert the identity of the user, as found in the CP token passed long with the request from Tomcat. This is available in the **session:current-request-info** attribute, via the **GetClientPrincipal()** method. With this handle available, we can pass it to a method that will handle all of the security for us, via the **SessionManager:establishRequestEnvironment()** method.

```
/* Obtain the CP token and establish user identity for session. */
define variable hCPO as handle no-undo.
assign hCPO = session:current-request-info
                    :GetClientPrincipal() no-error.
if valid-handle(hCPO) then
    Ccs.Common.Application:SessionManager
                          :establishRequestEnvironment(hCPO).
```

With this placed at the start of our WebHandler's HandleGet, HandlePost, etc. methods, we now know who is making the request for data and have secured our application session from the Tomcat server all the way to the database. We can then access PMFO features such as the **CurrentClientContext** object to obtain information about the current user.

```
/* Sample code to obtain the session ID via PMFO helper class. */
define variable oClientContext as IClientContext no-undo.
assign oClientContext = cast(Ccs.Common.Application:SessionManager
                             :CurrentClientContext, IClientContext).
if valid-object(oClientContext) then
    message substitute("SessionID: &1", oClientContext:contextID).
```

At the end of each Handle* method, preferably within a FINALLY block, we need to do the opposite and downgrade the session to an unprivileged user. This is done simply enough by use of the **SessionManager:endRequestEnvironment()** method. We should also delete our CP handle as we are done with it at this point and do not want it remaining in memory.

```
if valid-handle(hCPO) then
    Ccs.Common.Application:SessionManager:endRequestEnvironment().
delete object hCPO no-error.
```

Capstone Exercise

At this point you should have all the basic skills and tools required to build your own modern application with PMFO. If you do not have your own source of data, you may utilize the "AutoEdge: TheFactory" database (if provided by your instructor) as an alternative to the Sports2000 database. This means you will need to set up the necessary components for your project from scratch, using the processes as learned from chapters 2-4 of this material.

1. If necessary, create a new database from provided structure and schema files.
2. If necessary, create a new server for your database on a port of your choosing.
3. Create a new PAS instance for use with your new application.
    a. Choose ports that do not overlap with an existing PAS instance.
    b. Name your instance in a way that makes sense to you.
4. Create a new PDSOE project (ABL Web App) and connect to your database.
    a. Remember that you will replace the default ABL Service with the PMFO handler.
5. Import the PMFO ABL libraries from the server framework, add to PROPATH.
6. Tailor the project as necessary, disabling any unnecessary transports.
7. Implement security, initially with Form-Local (remember to set domains/keys).
8. Add the server to your PDSOE workspace, and add your new project to the PAS instance.
9. Deploy the WebWizard to your new PAS instance, allowing you to create content.
10. Create the standard Login and Landing pages for your Single-Page Application.
11. Create at least 2-3 business entities from tables in your available database.
    a. Publish, and test your endpoints by use of the Catalog Viewer.
12. Create UI screens from templates for each of your exposed resources.
13. Attempt to upgrade to Form-OERealm security for your application.

NOTES: