Dev Artifacts

Implementation and User Guide

Kevin McMillen

TIBCO Software Professional Services

[1 Overview 3](#_Toc94524143)

[2 Setup 4](#_Toc94524144)

[2.1 Dependencies 4](#_Toc94524145)

[2.2 Create properties file 4](#_Toc94524146)

[2.3 Add User Services 9](#_Toc94524147)

[2.4 Add REST Services (optional) 9](#_Toc94524148)

[3 Execution 10](#_Toc94524149)

[3.1 Environment copy 10](#_Toc94524150)

[3.2 Importing XSD changes 10](#_Toc94524151)

[3.3 Export 11](#_Toc94524152)

[3.3.1 Environment Copy 11](#_Toc94524153)

[3.3.2 Download to local 11](#_Toc94524154)

[3.4 Import 11](#_Toc94524155)

[3.4.1 Environment copy 11](#_Toc94524156)

[3.4.2 Publish workflow models 12](#_Toc94524157)

[3.4.3 Replace mode 12](#_Toc94524158)

[3.4.4 Skip non-existing files 12](#_Toc94524159)

[3.4.5 Workflow models 12](#_Toc94524160)

[3.4.6 Creating data spaces and data sets 13](#_Toc94524161)

[3.5 REST service 13](#_Toc94524162)

[3.5.1 Endpoints 13](#_Toc94524163)

[3.5.2 Input 14](#_Toc94524164)

[3.6 Command line 14](#_Toc94524165)

[4 Best practices 15](#_Toc94524166)

[4.1 General practices 15](#_Toc94524167)

[4.2 Source control and synchronizing changes 16](#_Toc94524168)

[4.3 Deploying to another environment 17](#_Toc94524169)

[4.4 Alternative practices 18](#_Toc94524170)

[5 Multi-tenant 18](#_Toc94524171)

[5.1 Admin tenant 18](#_Toc94524172)

[5.2 Message templates 18](#_Toc94524173)

[5.3 Addons 19](#_Toc94524174)

[5.4 Properties file 19](#_Toc94524175)

[5.5 User service 21](#_Toc94524176)

[5.6 REST service 23](#_Toc94524177)

[5.7 Command line 23](#_Toc94524178)

[5.8 Best practices 24](#_Toc94524179)

# Overview

There are many EBX artifacts that need to be synchronized between different environments. Models (XSDs), JSPs, and Java class files can be transferred to other environments by building them into a war file. However, there are many other artifacts such as permissions, roles, workflow models, and others. Transferring all of these artifacts can be time-consuming and error-prone. It mostly involves exporting archives or XML files, and then importing them into the other environment. It can be difficult to keep track of what changed and what needs to be migrated, and to make sure you’re not unintentionally overwriting something in the new environment.

TIBCO Customer Success developed the Dev Artifacts service as a solution to this difficult process. It provides the following benefits:

* **Ease of use.** Rather than having multiple places within EBX that you need to export from or import into, there is one place to go to execute the export or import. It is fairly easy to teach someone to use the Dev Artifacts service even if the person is relatively unaware with where all the artifacts are located inside EBX, or what artifacts are required to transfer.
* **Visibility into changes that are made in EBX.** Archive files are binary files and therefore can’t be compared. The Dev Artifacts service uses plain-text files. Some files may be difficult to interpret outside of EBX, but often you can infer changes from them even when the exact structure isn’t obvious. Even if you can’t interpret them, there is benefit to knowing whether changes happened. With text files, you can use any comparison tool to compare the old exported file with the new one. (Note that archives are actually zip files, so in theory one could unzip them and compare the contents. However, that would not be recommended, and would not be easy to do in an automatic fashion.)
* **Better collaboration between team members.** Because the Dev Artifacts service uses text files, you can check these files into a source control system, just like you would source code. When someone else checks in artifacts, and you update from your source control system, you can pull in their changes and merge them with yours using the merge tool of your choice, then import them into EBX. When you’re ready to share with others, you can export your artifacts from EBX and check into source control.
* **Reduced risk of errors.** The Dev Artifacts process is fairly simple to follow, and if followed correctly, greatly reduces the chances of introducing errors. Such common errors can include forgetting to export/import certain artifacts or overwriting local changes because an import file contained changes you weren’t expecting.
* **Automatable.** There is a REST interface to the Dev Artifacts service, as well as a command-line interface. This makes it possible to create an automated script that launches the service. For example, it could be part of a packaging or deploy process.

# Setup

## Dependencies

* The Dev Artifacts service works with EBX version 5.9.x and later. Earlier versions of the service support earlier versions of EBX, but won’t have all the functionality described here. Where some functionality is specific to EBX5 or EBX6, it is noted in the documentation.
* For EBX5, the Dev Artifacts service requires a jar file called adminpermissionsutil.jar. Your server needs to be configured to include this jar. For EBX6, the jar should not be included because its functionality has been integrated into the codebase.
* The Dev Artifacts service also relies on some Apache libraries, so commons-lang, commons-cli, commons-codec, httpclient, and httpcore jars must be included.
* The Dev Artifacts service can only be executed by a user with the role Tech Admin. If that role does not exist, it must be created. The actual name of the role can be changed if desired by providing a property tech.admin.role in the EBX Properties file, but by default it’s called Tech Admin.

## Create properties file

Create a file called dev-artifacts.properties. By default, the Dev Artifacts service will look for it in your EBX Home directory. If you want to put it somewhere else, you need to specify a JVM parameter upon startup called dev.artifacts.properties. For example:

-Ddev.artifacts.properties=../my-module/properties/dev-artifacts.properties

You can also change the name of the file if you wish, as long as that startup parameter points to the file.

*Note: This section describes a single tenant project, which is the standard way of using Dev Artifacts. For multiple tenant projects, there are additional parameters required which are described in Section 5*.

The properties file should look like this:

*These are the folders where it can find the artifacts.*

artifactsFolder = ../my-module/artifacts

copyEnvironmentFolder = ../my-module/WebContent/copyEnv

*This specifies the line separator type to use when exporting the files. You should use a consistent separator for your project so regardless of what system everyone uses, the exported file always is formatted similarly. Note that when using a different line separator than your operating system’s default, the export needs to write out the file once as a temporary file, then convert it. Therefore, it will take more processing time. For import, it doesn’t make a difference which line separator is specified.*

# windows (\r\n) or unix (\n)

lineSeparator = windows

*Specifies a role that is authorized to execute the service. This is optional and further restricts whatever the permissions (UI & programmatic) specify, which is Tech Admin by default.*

authorizedRole = DA Admin

*Specifies whether to enable a link in the UI for export, so that artifacts can be downloaded as a zip file from the server.*

enableDownloadToLocal = false

*Defaults for boolean options. Can be changed at execution time.*

# Default options

defaultEnvironmentCopyExport = false

defaultEnvironmentCopyImport = false

defaultPublishWorkflowModels = true

defaultReplaceMode = true

defaultSkipNonExistingFiles = false

*defaultDownloadToLocal = false*

*Specifies whether to include the name of the data space in the artifacts for data sets, and whether to include the name of the data set in artifacts for tables. It will result in longer filenames, but will allow for specifying data sets named the same in multiple data spaces or tables named the same in multiple data sets.*

qualifyDataSetAndTableFileNames = false

*These are tables that you need to maintain data for. Typically, they are administrative tables used by EBX patterns, not actual user data. But they can be whatever data you want to maintain in source control. Note that while an asterisk can be used to specify all tables in a data set, this does not support regular expressions.*

#Data Space | Data Set | Table Path (or \* for all)

tablesForData = \

AdminDataSpace|AdminDataSet|\*

*This specifies whether data spaces should be created if they don’t exist.*

# Create specified data spaces if they don't exist

createDataSpaces = true

*These are data spaces that you need to maintain permissions for. You can optionally specify the label and owner for use if you’re automatically creating data spaces. By default, will assume all data spaces are created in semantic mode unless a final token of true is specified after the owner.*

#Data Space | Parent Data Space | Label | Owner | Relational

#Only Data Space need be specified if createDataSpaces is false

#If Parent blank, assumes Reference (except Reference itself which ignores parameter)

#Relational only needs to be specified if true

dataSpacesForPermissions = \

Reference||Master Data - Reference|RTech Admin, \

AdminDataSpace||Admin Data Space|RTechAdmin, \

ProductMasterDataSpace||Product Master Data Space|RTech Admin, \

WorkflowPermissions||Workflow Permissions Template|RTech Admin, \

ARelationalDataSpace||A Relational Data Space|RTech Admin|true

*This specifies if schemas should always be refreshed on import. If false, they will only be refreshed when data sets are created.*

# Always refresh schemas on import

refreshSchemas = false

*This specifies if data sets should be created if they don’t exist*

# Create specified data sets if they don't exist

createDataSets = true

*Whether to also apply data set permissions to children of the data space*

dataSetPermissionsInChildDataSpaces = true

*These are data sets that you need to maintain permissions for. The xsd only needs to be specified if Dev Artifacts is creating the data sets. Creation of embedded data sets is not supported by Dev Artifacts. If you have embedded models in your DMA, you must publish them manually.*

#Data Space | Data Set | Data Model XSD (optional)

dataSetsForPermissions = \

AdminDataSpace|AdminDataSet, \

ProductMasterDataSpace|ProductMasterDataSet|my-module:/WEB-INF/ebx/schemas/ProductDataModel.xsd

*Whether you want to maintain permissions data for the administration data sets. Often, default permissions are used and in that case, less artifacts will need to be captured if this is set to false.*

adminDataSetPermissions = false

*Whether you want to maintain directory data.*

directory = true

*Whether you want to maintain global permissions data.*

globalPermissions = true

*Whether you want to maintain views data. Only published views are captured.*

views = true

*Whether you want to maintain tasks data. The schedules are not maintained, only the tasks themselves.*

tasks = true

*Whether you want to maintain perspectives data.*

perspectives = true

*A name to give the EBX window on import. This is optional, but can be specified if it should be different in this environment from what is configured in the advanced perspective.*

perspectiveWindowName = EBX Dev

*Whether you want to maintain historization profiles.*

historizationProfiles = true

*Whether you want to maintain add-on registration info.*

addonRegistrations = false

*A predicate on the Users Roles table indicating which users to maintain when maintaining directory data. If empty, all users will be maintained. If you wish to maintain no users, but maintain other directory data, then this should be set to a predicate that deliberately won’t match anything.*

usersRolesPredicate = \

./role = 'Permissions User' \

or ./role = 'Test User'

*Whether you want to remove the built-in administrator role from any users being imported.*

removeAdministratorRole = true

*The workflow models that serve as master workflows and which sub-workflows belong to them. This is only needed if you have master workflows and you want to auto-publish.*

#Master workflow | Sub-workflow 1 | Sub-workflow 2 | etc...

#Any workflow not listed will be published stand-alone (with no master)

#If a workflow is used both as a sub-workflow and as its own stand-alone workflow, then it should also be listed here with no sub-workflows

masterWorkflowModels = \

MasterWF|SubWF1|SubWF2|SubWF3

*The workflow models that should never be published when auto-publishing.*

#Workflow models that can be imported and exported but we want to ignore when publishing.

#configuration workflow model is a built-in workflow model that is always ignored so doesn't need to be listed.

workflowModelsToNotPublish = \

DataContextTemplate

*Whether you want to maintain message templates*

messageTemplates = true

*Whether information from the relevant add-on should be processed (if the add-ons are installed).*

addonAdix = false

addonDama = false

addonDaqa = false *(EBX5 only)*

addonDmdv = false

addonDqid = false

addonHmfh = false *(EBX5 only)*

addonMame = false *(EBX6 only)*

addonRpfl = false *(EBX5 only)*

addonTese = false *(EBX5 only)*

*If the ADIX add-on is being used, the prefix to use to identify import/export preferences that should be included with the artifacts. If not specified, no preferences will be included. If specified, only those preferences whose names start with this prefix will be included.*

addonAdixPreferencePrefix= [MDM]

*If the DAMA add-on is being used, the prefix to use for the physical root path for the Drive records. This can vary between environments so will be removed from the value on export and replaced on import.*

addonDamaDrivePathPrefix = ../data/

## Add User Services

Register the user services in an implementation of ModuleRegistrationListener. (See the EBX documentation for information on how to register user services.) The code would look something like this:

aContext.registerUserService(new ImportDevArtifactsPropertiesFileDSDeclaration(moduleName));  
aContext.registerUserService(new ExportDevArtifactsPropertiesFileDSDeclaration(moduleName));

You can register the user service with a particular data set, but typically you would register it with the data spaces and execute them from the Data Spaces tab. The PS Library class PSModuleRegistrationListener already registers these services, so if you just need the standard behavior and you’re using that class, it is already handled.

## Add REST Services (optional)

If utilizing the REST service, then register the service in an implementation of ModuleRegistrationListener. (See the EBX documentation for information on how to register REST services.) The code would look something like this:

aContext.registerRESTApplication(DefaultDevArtifactsRESTApplication.class);  
The PS Library class PSModuleRegistrationListener already registers these services, so if you just need the standard behavior and you’re using that class, it is already handled.

See Section 3.5 for more information about the REST service.

# Execution

## Environment copy

The Dev Artifacts service has a mode called “environment copy”, which is intended to be used when transferring a solution to another environment, rather than for day to day activities. If selected on export, then in addition to the normal exported artifacts, there will also be artifacts exported to the folder specified as the “copyEnvironmentFolder” in dev-artifacts.properties. It is that folder alone that would need to be transferred to other environments. On import, it will ignore all other artifacts and only import those from that folder.

The environment copy artifacts are the same as the normal artifacts except they also include an archive of the DMA (Data Modeler Assistant) information. In that way, you can sync model changes in the UI between environments. Normally, you would have to import the XSDs to make sure the new environment is in sync. Another difference is that on export, environment copy mode will clear the contents before exporting.

You could decide to always use environment copy mode in your source development environments, but typically you wouldn’t do this because as other developers check in their XSDs, you want to potentially merge them with your changes and import them. If you import the DMA they checked in, then you would be overwriting your DMA with theirs, which wouldn’t include your changes. Also, the DMA archive is a binary file, so it would always show up as a change in a source control system, even if nothing changed.

Likewise, you could decide to never use environment copy mode, but you would then be responsible for keeping the DMA in the target environment in sync. The intention is to use environment copy only when you are ready to promote your solution to another environment. Typically, you would only transfer these to the target environment and only use environment copy import there.

**NOTE: For EBX6, currently the DMA artifacts do not include the data model extensions (such as custom search strategies). For the time being, these must be configured manually in the target environment, or via an xsd import.**

## Importing XSD changes

This isn’t directly related to the Dev Artifacts service, but it’s worth noting how changes to XSDs are coordinated with the DMA (Data Modeler Assistant). If a change is made to an XSD outside of EBX, you must import the XSD into your model in the UI in order to keep it up to date. Importing an XSD will replace whatever you have in the DMA, so it’s a user decision, not automatically done any time the XSD changes. There is no API for doing this import, so the Dev Artifacts service doesn’t manage this process. It is still required that the users keep their DMA and XSD in sync using this process.

The exception to this is during an environment copy. An environment copy will take the entire DMA and archive it, and then replace the entire DMA in the other environment. Therefore, after doing an environment copy import, it is not required to also import the XSDs that changed.

## Export

Select the Export Dev Artifacts service, which is typically found on the Data Spaces tab. This will display the export form.

Upon submitting, all of the artifacts that are configured in dev-artifacts.properties, as well as all of the workflow models, will be exported to the folders specified. The Dev Artifacts service always exports every specified artifact, since EBX has no knowledge of what the state was at the time of the last export. You can compare it against what is in source control and know whether you have a difference from what is in your environment. The timestamp of the file may therefore show that it is new, but if doing a text compare, it will not detect any other change unless the actual content changed.

### Environment Copy

You have the option to select “Environment Copy”. This will cause the “environment copy” artifacts to also be exported, in addition to the normal artifacts. Typically, you would only select this option when you are ready to migrate to another server.

### Download to local

You have the option to select “Download to local”. This option is only shown when enabled in the properties file. When selected, it will cause the artifacts to be exported to the system’s temp folder rather than the typical location specified in the properties file. Then they will be zipped up and a hyperlink to that zip file will be displayed, allowing the user to download the file to their local file system via their browser. Since the exported artifacts and zip file are in the temp folder, they are subject to being reclaimed by the operating system and are not intended for long-term storage. Typically, development would occur in local environments and then promoted to remote servers, and therefore such functionality wouldn’t be useful. However, in situations where you wish to retrieve artifacts from a remote server, either to update a local environment or to perform analysis, this can provide a convenient way to do so, without requiring remote file access.

## Import

Select the Import Dev Artifacts service, which is typically found on the Data Spaces tab. This will display the import form.

Upon submitting, all of the artifacts that are configured in dev-artifacts.properties, as well as the specified workflow models, will be imported from the folders specified. Details of how the artifacts are imported depend on the choices the user makes in the form.

### Environment copy

You have the option to select “Environment Copy”. This will cause the “environment copy” artifacts to be imported. All other artifacts will be ignored. That’s indicated visually by all of the non-relevant checkboxes on the screen becoming hidden when this is selected. All of the “environment copy” artifacts will be imported in replace mode, thus completely replacing whatever changes you may have made in this environment. Typically, you would only select this option when you are migrating the artifacts from another server, and not when you are in an environment that you are doing active development on.

### Publish workflow models

If “Publish workflow models” is selected, then after import of the workflow models is complete, EBX will publish the workflow models that have changed since the last publication. If there are master workflow models, then they need to be indicated in the dev-artifacts.properties file so that they will be properly published along with their sub-workflow models. Otherwise it would be assumed that each workflow model corresponds to its own publication. (Sub-workflow models shouldn’t be published themselves unless they can also be executed independent from the master workflow model.) “Publish workflow models” is available regardless of whether or not you are in “copy environment” mode.

There is one scenario that is not covered automatically by the “publish workflow models” feature. If any message template changes, EBX requires that you publish all workflow models referencing that template before it will be reflected in the workflow. This means that on import, if a template changed, but the workflow models referencing it did not, then you will be required to manually publish those workflow models. Dev Artifacts isn’t capable of determining which workflow models need to be published based solely on templates changing.

### Replace mode

If “Replace Mode” is not selected, then EBX will import artifacts in an “insert/update” mode. This means it will insert new records or update existing records, but it won’t delete records if they don’t exist in the artifacts file being imported. If replace mode is selected, then records not existing in the artifacts file will be deleted.

For example, if you have a role in your environment that is not in the artifacts file, it will not be removed when replace mode is not selected, and it will be removed if replace mode is selected. (See the EBX documentation on import modes for more details.)

Replace mode always imports all workflow models found in the workflow models folder specified by the dev-artifacts.properties file. This is indicated visually by hiding all of the workflow model checkboxes.

### Skip non-existing files

When importing artifacts, the Dev Artifacts service will attempt to import all artifacts specified in the dev-artifacts.properties file. If one of the files it is expecting to find is missing, then an error will be recorded. Typically, this is desired because you want to make sure you aren’t missing an artifact, and if the artifact is missing, then it should not have been specified in dev-artifacts.properties. However, if you are aware of this and want to simply ignore this error, you can select “Skip non-existing files”. This will not consider it an error when expected files are not found.

### Workflow models

The import form will read the files in the workflows folder specified by the dev-artifacts.properties file, and create a checkbox for each one. (The filenames must match the expected format for the workflow model files: WF\_ + <workflow model name> + .xml. All workflow models that were exported via the Dev Artifacts service will match this format.) If the workflow is found in EBX, the checkbox will display its label. Otherwise, it will display the text (New) next to its name.

A workflow model is always imported in replace mode, since it needs to be treated as one entity. If a node of a workflow model was removed then it should be removed when importing. Otherwise the very behavior of the workflow model would be wrong, or the workflow model could be corrupted and not execute correctly at all. Therefore, if replace mode is not selected, you need to explicitly select the workflow models that you want to import to make sure you don’t overwrite workflow models that you may be working on in that environment.

The labels, descriptions, and owners of workflow models are captured in a file named the same as the XML file, but with a .properties extension. Upon import, the information of the workflow model will be updated.

After importing workflow models, if “Publish workflow models” wasn’t selected, then it is a manual process to publish whatever workflow models require publishing.

### Creating data spaces and data sets

When “createDataSpaces” is true in dev-artifacts.properties, specified data spaces that don’t exist upon import will be created.

When “createDataSets” is true in dev-artifacts.properties, and the XSD locations are specified, data sets that don’t exist upon import will be created. The format for specifying the XSD location is <module>:<xsd>. For example: “my-module:/WEB-INF/ebx/schemas/my-model.xsd”. If there are child data sets that need to be created, they will have to be handled manually.

The owner and label of a data set will be updated upon import if a file is found with the same name as the data set properties XML file, but with a .properties extension. This will work in a similar manner to how a workflow model label and owner gets set.

## REST service

Dev Artifacts can be invoked via a REST call. This can be used to integrate with packaging and deployment processes. It is built using the EBX REST Toolkit and therefore uses the standard mechanisms for invoking REST calls in EBX. (See the EBX documentation for information about the REST Toolkit.) There is a single REST service that handles both exports and imports. See Section 2.4 for information on registering the REST service.

### Endpoints

To invoke the REST service for an export, you would make a POST call to the following URL:

<ebx-host>:<ebx-port>/<module>/rest/v1/ps/devartifacts/export/execute

To invoke the REST service for an import, it is the same URL except substitute “import” for “export”:

<ebx-host>:<ebx-port>/<module>/rest/v1/ps/devartifacts/import/execute

The body of the REST call would be supplied as JSON. See Section 3.5.2.

There are also GET endpoints available that can be used for convenience, if using all default options. They are the equivalent of calling the POST endpoints with empty input ({ }).

<ebx-host>:<ebx-port>/<module>/rest/v1/ps/devartifacts/export/executeDefault

<ebx-host>:<ebx-port>/<module>/rest/v1/ps/devartifacts/import/executeDefault

### Input

The same options that would normally be configured via the user interface are available as input to the REST service, in the JSON body of the POST call. See the documentation of the user interface for more information about these options. They can be specified as fields named:

* environmentCopy (boolean)
* replaceMode (boolean)
* skipNonExistingFiles (boolean)
* publishWorkflowModels (boolean)
* workflowModels (list of strings)

If a field is not supplied, it will behave similar to the user interface when nothing is changed on the screen and Submit is clicked. The boolean options will default to the values supplied by the “default” properties in the properties file. i.e. If replaceMode is not supplied, and defaultReplaceMode is true, then Dev Artifacts will consider it true. As with the user interface, if replaceMode is true, then all workflow models will be processed and don’t need to be specified.

Here is an example of JSON input supplying values for all of the available fields:

{  
 “environmentCopy”: false,  
 “replaceMode”: false,  
 “skipNonExistingFiles”: false,  
 “publishWorkflowModels”: true,  
 “workflowModels” : [ “My\_WF1”, “My\_WF2” ]  
}

## Command line

Dev Artifacts has a Main class that can be invoked via the command line. It opens a connection to the specified Dev Artifacts REST Service endpoint and executes the service using Basic Authentication. The command line doesn’t provide any additional functionality that is not available via a direct REST service call, but could be useful for convenience or to integrate with a shell script.

All required libraries must be included in the classpath, including the EBX core and addons jars, and the third-party jars. At a minimum, the arguments that must be provided are the URL to either the export or import REST service POST endpoint, the username, and the password. To get the full usage, specify the help option (-h or --help) like so:

java com.orchestranetworks.ps.admin.devartifacts.main.DefaultDevArtifactsMain <host>:<port>/<module>/rest/v1/ps/devartifacts/export/execute <username> <password> -h

# Best practices

There are various ways that the Dev Artifacts service can be integrated into your development process. However, following these practices would be recommended for the typical project.

## General practices

* Only administrators should execute the Dev Artifacts services and their related servlets. Ensure that other users can’t access them.
* Be careful when you’re first getting used to the Dev Artifacts service. You can accidentally overwrite changes if you perform the wrong actions. Back up everything before launching the import until such time as you are comfortable with the process.
* If there are different operating systems being used with different line separator characters, then settle on which line separator will be used for the Dev Artifacts service. For example, if your local machine uses Windows but the integration server uses Linux, or if one developer uses Windows but another uses Mac, then you should all be using either Windows-style or Unix-style line separator to avoid seeing changes that are only the result of different line separator characters. (Note that XSDs will still use different line separators so you should ignore white space when comparing them.) If the line separator is different from your operating system’s default, then exporting will require writing out a temporary file and then converting it into another file. Therefore, the best choice may be a line separator that the majority of developers on the project are using, so that at least those developers don’t have to incur any extra processing time.
* The Dev Artifacts service lets you specify a predicate for the users that you want to maintain. This is because you will likely want different users in different environments, so you don’t necessarily want to migrate all of them. For example, you may want to maintain users used for testing purposes or used by certain programmatic processes, but not real users of the system. While you can specify various predicates to identify those users, the best solution is probably to give those users specific roles, such as a “Test User” role. Then your predicate can easily filter them by comparing the path ./role to “Test User”.
* Typically, the properties file will be the same in all environments, except for:
  + The folder paths will differ, since in a local environment, they are often expressed as relative paths to the folder inside the Eclipse workspace.
  + Default flags will sometimes differ, since in an environment where you’re not doing development, the most typical action will be to import with environment copy enabled, and that may not be most typical in the local environments.
  + Sometimes a different predicate will be specified for users in a production environment, so that test users don’t get imported there, but do in all other environments.

## Source control and synchronizing changes

* Store your artifacts in source control, as you would any other source artifact.
* Configure the folders for Dev Artifacts to be the same ones that are in source control, so that no manual copying from another location will need to be done.
* Store the dev-artifacts.properties file that you are using locally in source control, and configure your startup parameters so that you are pointing to that file (if it’s not in EBX Home). That way, when someone checks in an update to that file you will be using the latest version.
* Different servers may have different dev-artifacts.properties files. For example, the location of the artifacts may need to be specified differently on your local machine than on an integration server. If so, you may wish to store these variations in source control as well, and make it part of your deploy process to copy over these dev-artifacts.properties files if any change was made. Otherwise, that shouldn’t be necessary.
* Periodically export artifacts as you’re working so that you have a convenient backup if you should want to back out a mistake.
* You may need to refresh your IDE after exporting artifacts in order for updates to be reflected, since Dev Artifacts makes its changes directly to the file system and is agnostic about your IDE.
* To update your environment with changes others have made in source control:
  + Export your artifacts so that the file system has all of your changes.
  + Synchronize with source control.
  + Take note of any XSD and workflow model changes. You may wish to literally write down which changed so that you will remember later.
  + Review any differences to the artifacts files.
  + Update any artifacts that you know are not conflicting with your changes. Any changes to artifacts that you also made changes to should be reviewed even if no conflicts are detected and updated carefully.
  + Merge changes to the artifacts files. Many times, this is straightforward once you are familiar with the format of the artifacts files, but sometimes this can be difficult (such as if many changes were made to the same data set permissions). If that is the case, you may need to communicate with your team members to determine what changed and determine how to proceed.
  + Workflow models in particular can be difficult to merge, so it’s best if two people aren’t working on the same workflow model simultaneously.
  + Once the artifacts are merged on your file system, execute an import with “replace mode” and “publish workflow models” specified. Since the artifacts should now reflect all of your changes in addition to the changes from source control, then you can replace what’s in your environment from the artifacts.
  + If you believe that there are local changes that you don’t wish to be removed by the import, then you would not use replace mode. However, this would need to be done on a case-by-case basis and is most likely not what you would normally want to do.
  + Go to the Data Modeling tab and re-import any XSD files that you noted as changed. You should not have to republish if the data sets are linked directly to the module.
  + If you chose not to publish workflow models automatically, then you will need to go to the Workflow Modeling tab and republish any workflow models that were changed since your last publication. Note that if a subworkflow model changed, its parent workflow model must be republished for it to take effect. If the subworkflow model is only used as a subworkflow, then it need not be published itself at all.
  + At this point, your environment should be up to date. If you wish to make sure, you can execute an export. The exported artifacts should be identical to what is already on your file system (unless you didn’t use replace mode).
* To update source control with your latest changes:
  + Follow the above procedure for updating your environment from source control.
  + Check in your changes. Since the artifacts are not as easily interpreted as Java source code would be, and you can’t put comments in the artifacts files, it is especially valuable to enter comments describing what was changed.

## Deploying to another environment

* Make sure your environment is up to date and everything is checked in, as described above.
* Execute an export in “environment copy” mode. There should be no changes to artifacts other than the environment copy ones if everything was synced correctly before. If replace mode wasn’t used, then there could be changes but then you should not be deploying from this environment if there are changes you don’t want deployed.
* You may wish to check in the environment copy folder and tag it indicating the build. Or alternatively, you can choose to not check it into source control and tag the other artifacts, since those truly are the only artifacts that you need to reproduce the environment from.
* Transfer the environment copy folder to the server, replacing the entire existing environment copy folder if it exists. An alternate technique that is often used is to export the environment copy artifacts into the module file structure so that it is included in the generated war file that is deployed and point the dev-artifacts.properties for the remote system to that location. If this is done, you don’t need to separately copy the artifacts and it is more tightly associated with the deployed code.
* Transfer the module containing the compiled code, JSPs, and XSDs to the server. (How the module is built and transferred is not related to the Dev Artifacts service and can vary based on your development environment.)
* After restarting the server, execute an import in “environment copy” mode and specify “publish workflow models”.
* If for some reason, you don’t wish to automatically publish workflow models, then make sure to publish any workflow models that changed (or master workflow models of subworkflow models). If it’s unclear what changed, simply publish them all.

## Alternative practices

* You may wish to have a separate “clean” workspace used only for deploys if you find that you often have code or artifact changes that you’re not ready to deploy.
* You could not use the environment copy option at all and just import from the server in replace mode. However, you’d have to be diligent about importing the XSDs into the DMA. (That is normally done by the environment copy import.)
* Rather than transfer artifacts and the module to the server, you could set up your server to pull from source control directly. This depends on your infrastructure.
* You could utilize the REST service to automate exporting and importing of artifacts, possibly via a tool like curl. This is especially useful if being remotely invoked from another server.
* You could develop a script that utilizes the command line interface, which itself invokes the REST service, to automate exporting and importing of artifacts.

# Multi-tenant

The standard EBX installation uses a single module, containing all of the custom code and artifacts. A single development team is responsible for it, or if there are multiple teams, they coordinate with each other, and produce a single deliverable for deployment. That is the simplest approach, and therefore recommended. However, there are times that multiple development teams, or tenants, wish to operate independently although they share the same EBX instance. Each tenant would be responsible just for the artifacts related to their modules. In this case, Dev Artifacts can be operated in multi-tenant mode, which requires some additional configuration. A tenant can be responsible for multiple modules, but a module can only be associated to one tenant.

## Admin tenant

There are some artifacts that are common across all tenants or require coordination between tenants. These include built-in EBX artifacts that aren’t tenant-specific and often artifacts like common reference data sets, base perspectives that establish a common look for all tenants, workflow models to be used as templates, and others. For this purpose, one tenant needs to be specified as an “admin” tenant. This is a special designation that means it has more responsibility than a typical tenant. They can also operate as a standard tenant, or can be responsible for just the common artifacts.

## Message templates

Message templates have integer primary keys. Therefore, to specify which templates are managed by a tenant, an ID range must be established for the tenant by the admin. The tenant is responsible for making sure all of their message templates have IDs that fall within that range. Since the IDs are auto-generated, the tenant may need to manually edit the templates file to create a new template that is in the correct range, and then import and modify it after import, rather than create a new one in EBX. Another approach would be to create it in EBX and manually export it, then modify the ID in the file, import, and delete the original template.

## Addons

Currently, the add-ons with full multi-tenant support are: Data Exchange (ADIX), Digital Asset Manager (DAMA), Match and Cleanse (DAQA), Match and Merge (MAME), and Information Search (TESE). All other add-on data is considered part of the admin tenant. (DAQA and TESE are only available for EBX5. MAME is only available for EBX6.)

## Properties file

Each tenant has its own properties file. You need to specify a JVM parameter upon startup that points to that properties file. The parameter name can be whatever you want, as long as it’s unique. For example:

-Dmy-tenant.dev.artifacts.properties=../my-module/properties/my-tenant-dev-artifacts.properties

When registering the user service, you will configure it with the parameter name you specified here.

In addition to the standard properties, these additional properties should be specified:

*Specify multi for a standard tenant, or multi-admin for the admin tenant.*

# single (default), multi, multi-admin  
tenantPolicy = multi-admin

*The modules this tenant includes (comma-separated)*

modules = common-mdm

*In addition to view publications for the tenant’s modules’ models, also include view publications that start with the given prefix.*

tenantViewPublicationsPrefix = COMM\_

*In addition to view publications for the tenant’s modules’ models and those that start with the specified prefix, also include those listed here.*

tenantViewPublications = CommonInboxView,CommonActiveWorkflowsView

*Include perspectives that start with the given prefix.*

tenantPerspectivesPrefix = comm-

*In addition to perspectives that start with the specified prefix, also include those listed here.*

tenantPerspectives = ebx-perspective,common-perspective,

*A predicate on the Roles table, specifying which roles to include for this tenant that are potentially shared with other tenants. The users for these roles won’t be included with this tenant. This is most likely only useful for the multi-admin tenant.*

tenantSharedRolesPredicate = \

./name = 'Test User'

*A predicate on the Roles table, specifying which roles to include for this tenant, that aren’t shared with any other tenant. The users specified by usersRolesPredicate will be further restricted to only users that are in these roles.*

tenantRolesPredicate = \

starts-with(./name, 'COMM -')

*The range of IDs to use for message templates for this tenant, to ensure uniqueness across tenants.*

# startID-endID, inclusive

tenantMessageTemplateRange = 1-100

*A prefix that identifies which records from the ADIX add-on tables to include for this tenant, when ADIX is enabled. Pre-installed records that use the [ON] prefix are always included with the admin tenant.*

addonAdixTenantPrefix = [COMM]

*A prefix that identifies which records from the DAMA add-on tables to include for this tenant, when DAMA is enabled. Pre-installed records that use the [ON] prefix are always included with the admin tenant. Records in the Digital asset component table are determined from the specified data model.*

addonDamaTenantPrefix = [COMM]

*A prefix that identifies which records from the DAQA add-on tables to include for this tenant, when DAQA is enabled. Pre-installed records that use the [ON] prefix are always included with the admin tenant.*

addonDaqaTenantPrefix = [COMM] *(EBX5 only)*

*A prefix that identifies which records from the TESE add-on tables to include for this tenant, when TESE is enabled. Pre-installed records that use the [ON] prefix are always included with the admin tenant.*

addonTeseTenantPrefix = [COMM] *(EBX5 only)*

*A prefix that identifies which records from the MAME add-on tables to include for this tenant, when MAME is enabled. This is not relevant to those records that are associated with a data model since the module for those can be determined without using a prefix.*

addonMameTenantPrefix = [COMM] *(EBX6 only)*

## User service

Typically, there would be one user service registered for import and one user service registered for export. If using the PS Library, this is handled already, unless you need some non-standard behavior. However, with multi-tenant, each tenant needs their own set of user services. This way, the user can execute Dev Artifacts for a specific tenant. Since each tenant has its own properties file, its service needs to be configured to know which properties file to use.

As with single tenant mode, you register the service in an implementation of ModuleRegistrationListener, but in addition to specifying the module name, you would specify additional parameters in order to give the service a unique key and label, and to specify the properties file. The PS Library has some classes that help with this which are discussed below, but if you were to code this yourself, it would look something like this:

ExportDevArtifactsPropertiesFileDSDeclaration exportDevArtifactsServiceDeclaration = new ExportDevArtifactsPropertiesFileDSDeclaration(moduleName, exportServiceKey, exportTitle);  
exportDevArtifactsServiceDeclaration.setPropertiesFileSystemProperty(systemProperty);  
context.registerUserService(exportDevArtifactsServiceDeclaration);  
  
ImportDevArtifactsPropertiesFileDSDeclaration importDevArtifactsServiceDeclaration = new ImportDevArtifactsPropertiesFileDSDeclaration(moduleName, importServiceKey, importTitle);  
importDevArtifactsServiceDeclaration.setPropertiesFileSystemProperty(systemProperty);  
context.registerUserService(importDevArtifactsServiceDeclaration);

Where:

* [import/export]ServiceKey is a unique key for the service (e.g. “mytenant\_ExportDevArtifactsDS”).
* [import/export]Title is the string to appear in the menu (e.g. “[My Tenant] Export Dev Artifacts”).
* systemProperty is the name of the JVM system property that specifies the tenant’s properties file (e.g. “my-tenant.dev.artifacts.properties”).

The PS Library class PSModuleRegistrationListener registers the default single tenant version of these services, so if utilizing the PS Library, typically you would extend that class. However, with multi-tenant, you don’t want the standard Dev Artifacts services. If you wish to register other standard PS Library services, then you will still need to extend this class, but only one admin tenant module should do so. Otherwise, you would have multiple modules registering the same services. When extending the class, you would need to un-register the standard Dev Artifacts services and instead register your tenant’s services.

For this purpose, an extension of PSModuleRegistration is available, called PSMultiTenantAdminModuleRegistration. You can extend PSModuleRegistrationListener for your admin tenant and pass into the super constructor an instance of PSMultiTenantAdminModuleRegistration with the parameters for your tenant. For example:

@WebListener  
public class MyTenantModuleRegistrationListener extends PSModuleRegistrationListener  
{  
 public MyTenantModuleRegistrationListener()  
 {  
 super(new MyTenantModuleRegistration());  
 }  
}

where MyTenantModuleRegistration is a subclass of PSMultiTenantAdminModuleRegistration like so:

public class MyTenantModuleRegistration extends PSMultiTenantAdminModuleRegistration  
{  
 public MyTenantModuleRegistration()  
 {  
 super(  
 “my-module”,  
 “mytenant”,  
 “My Tenant”,  
 “my-tenant”);  
 }  
}

The result would be the same as the example above, except there is much less custom code.

The other non-admin tenants must not extend PSMultiTenantAdminModuleRegistrationListener and instead can extend PSMultiTenantModuleRegistrationListener like so:

public class MyTenantModuleRegistrationListener extends PSMultiTenantModuleRegistrationListener  
{  
 public MyTenantModuleRegistrationListener()  
 {  
 super(  
 “my-module”,  
 “mytenant”,  
 “My Tenant”,  
 “my-tenant”);  
 }  
}

## REST service

When calling the REST service for multi-tenant mode, you must specify the tenant’s JVM startup parameter name for the properties file. This is similar to how it’s passed into the user service declaration (see Section 5.5). In addition to the standard JSON input fields, you would also supply the following field:

* propertiesFileSystemProperty (String)

As always, the other fields are optional, if you wish for the default behavior. For example, to specify just environment copy mode and the properties file system property, and keep the defaults for everything else, the input would look like this:

{  
 “environmentCopy” : true,  
 “propertiesFileSystemProperty” : “my-tenant.dev.artifacts.properties”  
}

Since the properties file system property parameter is required at a minimum, the GET endpoints aren’t valid for multi-tenant mode.

## Command line

When invoking the command line for multi-tenant mode, you would use the same Main class as single tenant mode, but must make sure to specify the tenant’s JVM startup parameter name for the properties file. For example:

java com.orchestranetworks.ps.admin.devartifacts.main.DefaultDevArtifactsMain <host>:<port>/<module>/rest/v1/ps/devartifacts/export/execute <username> <password> -f my-tenant.dev.artifacts.properties

As with single tenant mode, you can use the help option (-h or --help) to get the full usage.

## Best practices

The best practices for multi-tenant mode are mostly the same as they are for single mode, with the exception that there are multiple projects now that must coordinate with each other.

* Each tenant will be dependent on the admin tenant. In addition, there could be dependencies between non-admin tenants.
* In the local environments of each tenant developer, they will need to import the artifacts from any other tenant that they are dependent on, including the admin tenant.
* The admin tenant would typically have every tenant in their local environment and ensure that tenants don’t conflict with each other. For example, make sure they don’t use workflow names that are the same as those chosen by other tenants. Establishing unique prefixes for tenants helps with this.
* Since the properties file is what determines which artifacts are included in a tenant, there’s nothing to stop one tenant from doing something that interferes with another tenant. The admin tenant may wish to be responsible for reviewing all properties files changes prior to them being promoted for other tenants’ use or deployed to a shared environment.
* The admin tenant should choose ranges for message templates that are large enough to accommodate potential growth, because once tenants start using a range, it can be hard to get everyone to renumber them.
* By default, any user that has the role Tech Admin can execute any tenant’s Dev Artifacts service. If you wish to have tenant-specific Tech Admin roles, you can create sub-roles of Tech Admin. Then, in each tenant’s properties file, you can specify the authorizedRole to be that tenant’s Tech Admin role. In addition, you can call setAuthorizedRole on the user service declaration object in the module listener to ensure that the service isn’t displayed in the UI unless you have that role.