QAD Adaptive Applications

Enterprise Edition

QAD Packaging Guide

70-3383-1.1
QAD Adaptive Applications
Enterprise Edition
February 2020

QAD Inc.
100 Innovation Place
Santa Barbara, California 93108
Phone (805) 566-6000
http://www.qad.com

Table of Contents

QAD Packaging Guide

This guide is used to create installations for QAD Adaptive ERP environments managed by YAB, including QAD Enterprise Edition and QAD Adaptive UX. The target audience for this guide is customers and partners who are looking to replace time-consuming and error-prone manual installation instructions with an automated solution that produces consistent results efficiently. The guide should be read in conjunction with the *Enterprise Edition Configuration and Administration Guide* and provides supplementary material for install developers. Developing an install solution touches on a wide range of topics and each solution has unique requirements. If you need further assistance please post your questions to the YAB Help forum:

https://groups.google.com/a/qad.com/forum/#!forum/yab-help

Product Name Changes

Starting in September 2019, the name for QAD's complete portfolio of products is QAD Adaptive Applications. Additionally, QAD Adaptive ERP is the new name for QAD's flagship ERP solution. QAD Adaptive ERP includes the functionality previously associated with QAD Cloud ERP and QAD Enterprise Applications - Enterprise Edition, plus the QAD Enterprise Platform and Adaptive UX, which resulted from the Channel Islands program. Going forward, the terms QAD Enterprise Applications, QAD Cloud ERP, and Channel Islands will be deprecated but will remain in previous documentation and training materials.

Introduction

Software is installed using the *install* command. The *install* command installs *packages*, where a package is a ZIP archive that associates the files in the archive with product metadata. A package is identified by a *name* and a *version*. The *name* i s also referred to as the *package class* in certain contexts. Administrators can list the packages installed in an environment using the *info* command. Install developers create packages using the *system-package-create* command.

See: *Packages* in the *EE Configuration and Administration Guide*

Preparing Content

A package is a means to control the distribution of files. After the package is installed into an environment, the package files are available in the local catalog:

```
> yab install example-1.0.0.0.zip
> yab config packages.example.dir
.../build/catalog/packages/example/1/0/0/0
```

Most packages will also change the environment into which they are installed and these changes are ultimately effected by changing the configuration of the environment. There are two main ways configuration changes are introduced into the environment:

1. *Directly* - By defining new configuration settings or updating existing configuration settings.
2. *Indirectly* - By distributing files that result in the system producing new configuration settings.

The direct method is exemplified by the distribution of YAB configuration files and the indirect method by distributing content (e.g. schema, data, code, etc) that is discovered by the system resulting in new configuration to deploy the content. Many packages use both approaches, leveraging content discovery to easily and reliably set up the deployment of various artifacts and then direct configuration to fine tune the results as necessary.

A wide range of effects can be described through configuration. Configuration can be used to:

- Define new servers and components (new database, application server, web application).
- Reconfigure standard commands (process additional schema and data, compile code, deploy client plugins).
- Define new custom commands.
- Schedule custom commands to run before or after standard commands.

Many of QAD's standard packages also use direct configuration and indirect content discovery but some also rely on support from YAB. For example, the `yab-css` module provides deployment support for the CSS add-on. These packages are usually not the best reference model for the material in this guide. The following command will show the directories that are being scanned for content and is useful to see examples of how various content is distributed using the techniques outlined in this guide.

```
yab -r -v -log-level:trace | grep "FbcScanProcess - Scanning"
```

Configuration Files

YAB is configured through files distributed in the following package directory:

```
yab/config
```

The directory can contain multiple files with any name as long as the files end with the extension ".properties". Configuration settings are defined using a "property file" syntax:

KEY=VALUE

Lines that begin with a "#" are processed as comments or annotations. The value of other settings can be referenced using the syntax "${KEY}" and this includes settings defined before and after the setting and in other files and packages. The settings defined in a package are integrated into the environment as "factory default" settings. Administrators have the ability to override the settings.

Package Location

When a package is installed into an environment, the system defines settings that can be referenced by package authors in their configuration files. One of the most important settings configures the location of the package:

`packages.[CLASS].dir`

Where CLASS is the name given to the package when it is created. For example, the location of a package with the class "example" would be located with the following system setting:

`packages.example.dir`

When content discovery is enabled for a package that defines branches, the system will generate a setting to locate the directory in which the evaluated package has been staged:

`packages.[CLASS].staged`

The location of the package can be used to reference resources distributed in the package, as in the following example, where the system is configured to load data serialized in XML format (see content discovery for an easier way to set up data loading):

```
data.example.dir=${packages.example.dir}/data
data.example.includes=*.xml
data.example.format=xml
data.example.database=db.qaddb
```

Package Version

The version of a package is defined with the system maintained setting:

`packages.[CLASS].version`

The version is often the subject of conditions (which use the if or unless annotations to test configuration settings) to conditionally define configuration settings as in the following example:

```
# @if packages.fin-bin64-qadfin =* [2017.1)
data.example.dir=${packages.example.dir}/data
data.example.includes=*.xml
data.example.format=xml
data.example.database=db.qaddb
# @end
```

YAB uses version ranges to identify a contiguous sequence of versions with a syntax based on "interval notation".

Version Range	Interpretation
1.2.3.4	Version 1.2.3.4
[2017.1)	Version >= 2017.1
[2017.1]	Version >= 2017.1
(2017.1]	Version > 2017.1
[1.2,1.3)	Version >= 1.2 and less than 1.3

Configuration Types

There are many different actions that can be configured. The `config-help` command is used to get help on different configuration types. For example, instances of the *process* configuration type are used to configure new commands. To find more about the *process* configuration type use the following command:

```
> yab config-help process
```

One of the best ways to understand what is available and how it is done, apart from the help, is to look at examples of configuration. The `config` command prints configuration settings. The (-trace) option and the (-skip-resolution) option are useful to see how settings are defined prior to their final evaluation.

See Also

Configuration Files (*EE Configuration and Administration Guide)*

Configuring Commands (*EE Configuration and Administration Guide)*

Configuration Type System (EE Configuration and Administration Guide)

TCP/IP Ports (EE Configuration and Administration Guide)

New Servers

An important use for configuration is to define new servers. For example, a new Progress database and database server can be configured through instances of the "db" and "dbserver" type sharing the same instance name. In the following example, an "example" database and then a database server for that database are defined, both using the instance name "example":

```
# @extends db._base
db.example=
db.example.physicalname=example
```

```
# @extends dbserver._base
dbserver.example=
dbserver.example.name=db-${db.example.physicalname}
dbserver.example.databasename=${db.example.dir}/${db.example.physicalname}
```

In recent versions of YAB, the `system-config-new` command provides a quick way to generate a minimal set of configuration for a new instance:

```
> yab system-config-new db example

# @extends db._base
db.example=
db.example.physicalname=example
```

The "@extends" annotation is used to inherit from a base configuration. The settings that are inherited can be reviewed with the `config` command and overridden as desired. Use the `config-help` command to list all the options for databases and database servers:

```
> yab config-help db
> yab config-help dbserver
```

YAB uses this configuration to create and configure the database during the install and to provide the administrator with commands to manage the database.

```
yab command-help database-example

PROCESSES

     database-example-ai-archiver-disable                Disables the AI archiver for the
'example' database.
     database-example-ai-archiver-enable                 Enables the AI archiver for the
'example' database.
     database-example-ai-archiver-start                  Starts the AI archiver daemon for the
'example' database.
     database-example-ai-archiver-stop                   Stops the AI archiver daemon for the
'example' database.
     database-example-ai-disable                         Disables AI on the 'example' database.
     database-example-ai-enable                          Enables AI on the 'example' database.
     database-example-ai-list                            Lists AI extents on the 'example'
database.
     database-example-ai-status                          Checks whether AI is enabled for the
'example' database.
     database-example-ai-switch                          Switches to the next AI extent on the
'example' database.
     database-example-auditing-archive                   Moves audit records in the 'example'
database to the archive database.
     database-example-auditing-disable                   Disables auditing on the 'example'
database.
     database-example-auditing-enable                    Enables auditing on the 'example'
database.
     database-example-backup                             Creates a backup of the 'example'
database.
     database-example-backup-list                        Lists the backup tags containing
backups of the example database.
     database-example-backup-mark                        Marks 'example' database as backedup.
```

```
     database-example-configure
     database-example-create                              Creates the 'example' database.
     database-example-data-dotd-update                    Loads DOTD format data into the
'example' database.
     database-example-data-update                         Loads data into the 'example' database.
     database-example-data-xml-update                     Loads XML format data into the
'example' database.
     database-example-dataload-start-stop                 Ensures the 'example' database is
running when data is loaded.
     database-example-datastore-create                    Creates a datastore in the 'example'
database.
     database-example-describe                            Displays descriptive information about
the 'example' database and enabled features.
     database-example-index-deactivate                    Perform an index deactivate.
     database-example-index-rebuild                       Perform an index rebuild.
     database-example-log-print                           Prints the 'example' log.
     database-example-log-truncate                        Truncates the log of the 'example'
database.
     database-example-rebuild                             Rebuilds the 'example' database.
     database-example-remove                              Removes the 'example' database.
     database-example-restart                             Restart the 'example' database
(Restart with cautions since other services may depend on the server).
     database-example-restore                             Restores a backup of the 'example'
database.
     database-example-schema-locate                       Locates schema files defining tables
and sequences for the 'example' database.
     database-example-schema-snapshot
     database-example-schema-update                       Applies schema to the 'example'
database.
     database-example-security-authentication-update      Turns on/off authentication for the
'example' databases
     database-example-security-reapply                    Applies security configuration after
restoring a backup.
     database-example-security-table-permission-update    Updates table permissions for the
'example' databases
     database-example-security-update                     Updates the security information for
'example' database.
     database-example-security-update-online              Updates the security information for
'example' database while it is online.
     database-example-sequence-info                       Prints sequence names and (optionally)
the current value of the sequences.
     database-example-start                               Starts the 'example' database.
     database-example-status                              Checks the status of the 'example'
database.
     database-example-stop                                Stops the 'example' database.
     database-example-storage-area-resolve                Resolve the storage area
inconsistencies for 'example' database.
     database-example-structure-file-update               Generates the default structure for
the 'example' database.
     database-example-structure-file-validate
     database-example-structure-list                      Prints the structure of the 'example'
database.
     database-example-structure-update                    Applies new areas to the 'example'
database.
     database-example-structure-validate                  Validates the structure configuration
for the 'example' database.
     database-example-table-info                          Prints table names and (optionally)
the number of records they contain.
     database-example-truncate                            Truncates the before image of the
'example' database.
     database-example-update                              Updates the 'example' database.
     database-example-users                               Lists users connected to the 'example'
database.
     database-example-users-dba-update                    Grants/revokes 'dba' and 'resource'
privilege in the 'example' database.
     database-example-users-update                        Creates/updates database users in the
'example' database.
```

See Also

Adding Custom Database (*EE Configuration and Administration Guide)*

Adding Custom Application Server (EE Configuration and Administration Guide)

Content Discovery

Content discovery is an approach that analyzes the layout of a package to find the content that the package distributes and generates an appropriate configuration to deploy the content.

To enable content discovery, define a new instance of the `scan` configuration type in a package configuration file.

Ex. Configure the system to scan the 'example' package for content starting from the package root directory:

yab/config/example.properties

```
scan.example.dir=${packages.example.dir}
scan.example.enabled=true
```

The help provides additional details on less frequently used settings:

```
> yab config-help scan
```

See Also:

Configuration Type System (*EE Configuration and Administration Guide)*

Layout

YAB is continually adding support for the automatic discovery and deployment of new types of content. The best way to understand what is supported in a given version of YAB is to review the content discovery *profiles* that are used to match content in that version. Profiles are JSON format documents configured by the `fbc.scan.profiles` setting:

```
> yab config fbc.scan.profiles
fbc.scan.profiles=/dr01/qadapps/qea/build/catalog/packages/yab-ee-foundation/1/9/0/17/etc/fbc
/foundation.profile
```

The "type" value is a code that identifies the type of artifact and the "includes" are file patterns to match content defined relative to the directory that is scanned (typically the package root). For example, the following rule will identify any XML file (file with a ".xml" extension) in the directory "browse-collection" or "ui/browse-collection" as a Browse Collection (a type of .NET UI document) to be deployed during an `update` (or more specifically during the `browse-collection-update` step in an `update`). Some rules (like this one) will have multiple patterns configured for backwards compatibility. In these cases, the first pattern describes the preferred location for the content.

```
{
    "type": "ee.browse-collection",
    "includes": ["browse-collection", "ui/browse-collection"],
    "matchIncludes": ["*.xml"]
}
```

Some rules use a "*" wildcard to match a directory name that can vary. For example, the following rule discovers XML formatted data to load into a Progress database.

```
{
    "type": "progress.data.xml",
    "instance": "FILENAME",
    "includes": ["data/progress/xml/*", "data/qaddb*", "data/xml/qaddb*", "data/qadadm*", "data
/xml/qadadm*", "data/qadhlp*", "data/xml/qadhlp*"],
    "matchIncludes": ["*.xml"]
}
```

The "*" matches a directory name that is used to identify the database into which the data should be loaded. In this case, the expectation is that the directory name is the configuration name of the database (db.qaddb, db.qadadm) without the "db" prefix (qaddb, qadadm). So, for example, a package could distribute data for the "qaddb" and "qadadm" database in the following package directories:

```
data/progress/xml/qaddb
data/progress/xml/qadadm
```

The (-instances) option of the `config` command is used to enumerate the instances of a type. In the following example, -instances lists the Progress databases in the environment.

```
> yab -instances config db
db.bisgen
db.extension
db.qadadm
db.qadarc
db.qadcpl
db.qaddb
db.qadeam
db.qadhlp
db.qadrcode
db.qxevents
db.qxodb
...
```

A related example, is the rule that locates Progress code to compile where the "*" matches the configuration name of a compile (code.mfg, code.qxtend) without the "code" prefix (mfg, qxtend).

```
{
    "type": "progress.code",
    "instance": "FILENAME",
    "includes": ["code/progress/*"]
}
```

So, for example, a package could distribute Progress code to be compiled by the operational compile (code.mfg) in the following package directory:

```
code/progress/mfg
```

The layout of the source code should match the desired layout of the r-code. So for example, to compile the operational program "example.p" to "us/xx/example.r", the program should be distributed as "code/progress/mfg/us/xx/example.p" in the package.

See Also

Compiling Source Code (EE Configuration and Administration Guide)

Branches

Branches are used to distribute content that is appropriate for some but not all target environments. A branch is associated with a Boolean expression that is evaluated in the target environment. If the expression evaluates to false the content in the branch will not be applied to the environment. Packages that define branches are rewritten into a staging directory after all of the branches have been evaluated:

```
[ENVIRONMENT]/build/work/stage/[CLASS]
```

The system defines a setting locating the directory in which the branched package was staged:

```
packages.[CLASS].staged
```

Branch Configuration

Branches are configured with settings associated with the `scan` configuration type. In the following example, two branches are configured "2016" and "2017". The "2016" branch is associated with an expression that evaluates to true when the version of the "mfg" package in the target environment is >= 2016 and the "2017" branch is associated with an expression that evaluates to true when the version of the "mfg" package in the target environment is >= 2017. Furthermore if both branches evaluate to true (i.e. in a 2017 environment), files in the "2017" branch should take precedence over files in the "2016" branch.

```
scan.example.dir=${packages.example.dir}
scan.example.enabled=true
scan.example.branches.usecollapsedbranches=false
scan.example.branches.precedence=2017,2016
scan.example.branches.2016=packages.mfg.version.isMember('[2016]')
scan.example.branches.2017=packages.mfg.version.isMember('[2017]')
```

The `usecollapsedbranches` setting should always be set to false. It is true by default for backward compatibility.

Branches can be used to redistribute different versions of programs that would be selectively compiled based on the version of the "mfg" package in the target environment:

```
[PACKAGE]/code/progress/mfg/us/wh/whexeng.p                    //compiled when mfg < 2016
[PACKAGE]/code/progress/mfg/us/wh/whexaud.p                    //compiled in all environments
[PACKAGE]/code/progress/mfg/branches/2016/us/wh/whexeng.p      //compiled when mfg >= 2016 and <
2017
[PACKAGE]/code/progress/mfg/branches/2017/us/wh/whexeng.p      //compiled when mfg >= 2017
```

The correct version of `whexeng.p` for the target environment is located in the staging directory:

```
[ENVIRONMENT]/build/work/stage/example/code/progress/mfg/us/wh/whexeng.p
```

Branch Expressions

Branch expressions support the following syntax:

```
[PROPERTY] [=,!=,>=,>,<=,<] [VALUE]
[VERSION PROPERTY].major [=,!=,>=,>,<=,<] [INTEGER]
[VERSION PROPERTY].minor [=,!=,>=,>,<=,<] [INTEGER]
[VERSION PROPERTY].revision [=,!=,>=,>,<=,<] [INTEGER]
[VERSION PROPERTY].subrevision [=,!=,>=,>,<=,<] [INTEGER]
[VERSION PROPERTY].isMember([VERSION RANGE])
```

Capturing Schema and Data

YAB provides developer-oriented commands to help manage schema and data.

File Commands

Some commands work at the level of individual files.

```
> yab command-help schema

PROCESSES

     schema-diff     Compares two schema files.
     schema-dump     Dumps schema from a Progress database.
```

```
> yab command-help data-xml

PROCESSES

     data-xml-diff       Compares XML format data files for differences.
     data-xml-dump       Dumps data from a Progress database in XML format.
     data-xml-format     Formats an XML data file.
     data-xml-index      Prints the ID of records in an XML format data file.
     data-xml-load       Loads data from a Progress dataset in XML format.
     data-xml-new        Creates an empty XML data file.
     data-xml-patch      Adds/removes records from a target data file.
```

Snapshot Commands

Other commands are based on a snapshot metaphor where a snapshot is taken before a change (capturing schema or data) and then a snapshot is taken after the change. The difference between the "before" and "after" snapshots can be used to maintain the schema and data that are distributed in the package.

```
> yab command-help dev

PROCESSES

     dev-data-clear             Deletes all data snapshots.
     dev-data-diff              Compares two snapshots.
     dev-data-entity-list       Lists the configured data entities.
     dev-data-list              Lists all data snapshots.
     dev-data-patch             Compares two snapshots merging differences into one or more data
files in a directory.
     dev-data-snapshot          Creates a new data snapshot.
     dev-data-snapshot-auto     Create an initial dev-data snapshot when the environment is
created.
     dev-data-snapshot-list     Lists all data snapshots.
     dev-package-remove         Removes developer packages from the environment.
     dev-schema-clear           Deletes all schema snapshots.
     dev-schema-diff            Compares two schema snapshots.
     dev-schema-list            Lists all schema snapshots.
     dev-schema-snapshot        Creates a new schema snapshot recording the schema of one or more
databases.
```

See Also:

Schema Management (EE Configuration and Administration Guide)

Data Management (EE Configuration and Administration Guide)

Packaging

A package is created using the `system-package-create` command, which accepts one or more paths locating content to include in the package. A standard practice is to organize all content within a single directory under revision control and supply that directory as the argument to the command.

Ex. Create the 'example-1.0.0.0.zip' package to redistribute the files within '/dr01/example'

```
> yab -class:example -version:1 system-package-create /dr01/example
```

The *class* must only contain the alphanumeric characters ('a-z', '0-9') and dashes ('-') and may not end with characters that could be interpreted as a version (e.g. "-[NUMBER]"). The *version* consists of a set of four integers in the range of 0 to 65535 from the most significant to the least significant value (when read from left to right). If any integer is omitted, the default is "0" (i.e. 1 = 1.0.0.0).

The help provides additional details on the command:

```
> yab command-help system-package-create
```

The package file that is created has the following format:

Entry	File	Description
1	metadata.xml	The package metadata including the package class, version, and any attributes defined.
2	data.zip	A nested ZIP that contains all of the package files.

See Also:

system-package-create (EE Configuration and Administration Guide)

Package Specification

A package specification is an XML document that provides additional options for customizing the metadata associated with a package and through that the user's experience when the package is installed. These are optional techniques and many packages will not need to define a package specification, but the following sections provide guidance on some common techniques.

The (-spec) option references the package specification when the package is built using the `system-package-create` command:

```
> yab -spec:package-specification.xml -class:example -version:1 system-package-create /dr01
/example
```

Defining the Product Name

The product name is displayed when the install is run interactively and the package has a user interface (see below). The product name is defined in a file named "install.inf" and requires configuration in the package specification to set up the contract. The following package specification is suitable for a file distributed as "etc/install.inf".

etc/install.inf

```
Example Product
```

package-specification.xml

```
<?xml version="1.0" encoding="utf-8"?>
<package-specification>
    <package-meta>
        <bundle-mapping>
            <filter bundle-path="/" flatten="true">
                <include>etc/install.inf</include>
            </filter>
        </bundle-mapping>
    </package-meta>
</package-specification>
```

User Interface

The user interface is displayed when the install is run interactively (default). The user interface is defined in a file named "install.conf" and requires configuration in the package specification to set up the contract. The following package specification is suitable for a file distributed as "etc/install.conf".

etc/install.conf

```
# [Work Directory]
#
# Enter the location of the work directory:
# @path
# @validation req
# @validation dir
example.work.dir=
```

package-specification.xml

```
<?xml version="1.0" encoding="utf-8"?>
<package-specification>
    <package-meta>
        <bundle-mapping>
            <filter bundle-path="/" flatten="true">
                <include>etc/install.conf</include>
            </filter>
        </bundle-mapping>
    </package-meta>
</package-specification>
```

The install user is prompted for the location of the work directory when the example product is installed. The system validates the entry to ensure it is a directory and then integrates the setting into "build/config/system/environment.properties" in the environment. The install user can run the installer non-interactively as long as they provide an acceptable value for the working directory.

Most settings associated with the package should be defined in YAB configuration files. These settings are integrated into the system as "factory defaults" and the user is free to selectively adjust them. Settings in the `install.conf` file by contrast are integrated into the system at a higher level of precedence and will override user configuration when installed. The `install.conf` should be used primarily as a means to define settings that are interactive and/or to define settings to configure the installation (i.e. settings that begin with "install-").

Another use of the `install.conf` is to define the command(s) that should be executed when the package is installed. The default is to run an `update`, which is suitable for many packages. This extended install.conf adds a "meta" configuration entry to instruct the installer to not prompt for settings below it and then instructs the installer to run the `example-update` command when the product is installed.

etc/install.conf

```
# [Work Directory]
#
# Enter the location of the work directory:
# @path
# @validation req
# @validation dir
example.work.dir=

# All entries below this line will not be displayed in the installer user interface.
# @hide-all
install-ui-end=

# The command that will be executed by the installer after the product is integrated.
install-command=example-update
```

Quarantining Settings

Quarantined settings are removed when the product is installed and the settings that were removed are logged to the file "build/config/quarantined.properties" in the environment. A quarantine file should enumerate the keys to remove with the equals sign (but the value can be omitted) and requires configuration in the package specification to set up the contract. The following package specification is suitable for a file distributed as "etc/example-quarantine.properties".

etc/example-quarantine.properties

```
example.working.dir=
```

package-specification.xml

```
<?xml version="1.0" encoding="utf-8"?>
<package-specification>
    <package-meta>
        <bundle-mapping>
            <filter bundle-path="etc/conf/quarantine" flatten="true">
                <include>etc/example-quarantine.properties</include>
            </filter>
        </bundle-mapping>
    </package-meta>
</package-specification>
```

Defining a Dependency

A dependency asserts a requirement on another package. A product that depends on features available in YAB 1.9 or greater can prevent the installation of the product into an inappropriate environment by defining a dependency in the package metadata.

package-specification.xml

```
<?xml version="1.0" encoding="utf-8"?>
<package-specification>
    <package-meta>
                <package>
            <runtime>
                <reference>
                    <depends>yab-ee-app[1.9]</depends>
                </reference>
                        </runtime>
                </package>
    </package-meta>
</package-specification>
```

Installing

A package is installed using the `install` command.

Ex. Install 'example-1.0.0.0.zip' into the environment.

```
> yab install example-1.0.0.0.zip
```

The installer will execute the `update` command by default after integrating the package into the environment, which will take the environment offline temporarily, but is the right choice for most products.

When developing an installer sometimes it is useful to skip the update using the (-install-update) option:

```
> yab -install-update:false install example-1.0.0.0.zip
```

After the product is installed it is listed by the `info` command:

```
yab info | grep example
|- example                                1.0.0.0        local
```

The help provides additional details on the *install* command:

```
> yab command-help install
```