QAD Enterprise Applications
Enterprise Edition

QAD Packaging Guide

70-3383-1.0

QAD Enterprise Applications

Enterprise Edition

September 2018

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 provides instructions for creating packages that can be integrated into QAD Enterprise Edition environments administered by YAB. The target audience for this guide is customers and partners who want to provide a reliable turnkey solution for integrating software products and/or configuration into Enterprise Edition environments.

The *EE Configuration and Administration Guide* provides additional information on many of the concepts described in this guide.

Introduction

A package is a ZIP archive with a proprietary format that associates the files in the archive with metadata. Every package has a *class* and a *version* and the combination of the class and version is used to identify the package. The class is a name for the software or solution being distributed and the version is used to distinguish between different releases of the software or solution. A package should be managed as a read-only resource to ensure that the version is a reliable way to identify a specific release.

The use of packages provides a controlled way to upgrade an environment and gives administrators visibility into their current product configuration with commands like `info` that list the enabled packages and their versions.

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

Preparing Content

There are no inherent limits restricting how files must be organized in a package, but there are layout conventions that are designed to reduce the effort it takes to create a working solution. Packages that distribute YAB configuration files in a well-known location are able to define new services, commands, and settings and enable content discovery to automatically discover and configure content distributed in the package.

Configuration Files

YAB is configured through property files (files with a ".properties" file extension) distributed in the following directory.

```
[PACKAGE]/yab/config
```

The settings defined in these property files are integrated as "factory default" settings that can be overridden /reconfigured by an administrator. Settings are defined using the following syntax:

KEY=VALUE

Lines that begin with a "#" are handled as comments or annotations. The value of other settings can be referenced using the syntax "${KEY}" and this includes settings defined in other files and packages.

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 locates 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 this up):

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

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
```

There are many different types of resources that can be configured and set up in an Enterprise Edition environment. Use the `config` command to review examples. Use `config` with the -trace option to see the original configuration settings. The `config-help` command provides help for the different types supported by the system. For example, instances of the `process` type (re)configure commands and the following command shows the available process options.

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

See Also

Configuration Files (*EE Configuration and Administration Guide)*

Configuring Commands (*EE Configuration and Administration Guide)*

Content Discovery

Content discovery is an approach where the layout of a package is analyzed to find the content it distributes and appropriate configuration is generated to deploy the content.

To enable content discovery define a new instance of the `scan` configuration type.

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

```
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

QAD is continually adding support for the discovery of new content types. The best way to understand what is supported in a given version of YAB is to review the content discovery *profiles* that define the rules that are used to match content. Profiles are JSON format documents located 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 root directory in a package). 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 surgically during a `browse-collection-update`). Some rules (like this one) will have multiple patterns configured for backwards compatibility. In these cases, the first pattern describes the preferred location.

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

Some rules use a "*" wildcard to match a variable directory, where typically the "*" matches a resource in the environment. For example, the following rule is used to load data in a database and the "*" matches a directory that identifies the database that is the target of the operation.

```
{
    "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"]
}
```

For example, a package could distribute data for the "qaddb" and "qadadm" database with the following package layout:

```
[PACKAGE]/data/progress/xml/qaddb
[PACKAGE]/data/progress/xml/qadadm
...
```

These names are derived from the YAB configuration names for databases (with the "db." prefix removed):

```
> 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
...
```

Another example, is the rule that locates Progress code to compile where the "*" matches a compile in the environment.

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

Progress programs that should be compiled with the operational compile (code.mfg) can simply be distributed in the following package directory:

```
[PACKAGE]/code/progress/mfg
```

```
> yab -instances config code
code.activity-feed
code.edi-default
code.fin
code.mfg
code.mfg-customizations
code.qxi
code.qxo
...
```

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 backwards 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 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])
```

A version range is used to describe a continuous range of versions using a mathematical interval notation.

Ex.

Version Range	Description
`[3.1, 3.2)`	Version is >= 3.1 and < 3.2 (i.e 3.1.x).
`[3.1]`	Version is >= 3.1.
`[0, 3.1)`	Version is < 3.1.
`[3.1, 3.1]`	Version is exactly 3.1.0.0

Developer Processes

Enterprise Edition environments include the same commands that QAD software developers use to capture Progress schema and data changes. The developer commands all begin with "dev" prefix. Also useful are the "lower level" commands for managing schema and data files.

```
yab command-help dev schema 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.
     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         Dumps the current state of all managed entities.
     dev-data-snapshot-auto    Create an initial dev-data snapshot when the
environment is created.
     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.
     schema-diff               Compares two schema files.
     schema-dump               Dumps schema from a Progress database.
```

See Also:

Schema Management (EE Configuration and Administration Guide)

Data Management (EE Configuration and Administration Guide)

Compiling Source Code (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. The following example demonstrates use of the *<bundle-mapping>* section to configure standard installation documents that define the name of the product that is displayed when the package is installed interactively (install.inf) and settings to configure the installation and/or prompt the user for input. These are discussed in greater depth in the Customizing Installation section.

package-specification.xml

```
<?xml version="1.0" encoding="utf-8"?>
<package-specification>
    <package-meta>
        <package>
            <class>example</class>
            <display-class>Example Product</display-class>
            <vendor>Example Products Inc</vendor>
        </package>
        <bundle-mapping>
            <filter bundle-path="/" flatten="true">
                <include>etc/install.inf</include>
                <include>etc/install.conf</include>
            </filter>
        </bundle-mapping>
    </package-meta>
</package-specification>
```

The package that corresponds to this specification would distribute the following files.

```
[PACKAGE]/etc/install.inf
[PACKAGE]/etc/install.conf
```

To build the package from the specification use the (-spec) option to reference the package specification file:

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

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. The Customizing Installation section describes how to configure a different command to apply the product.

The help provides additional details on the command:

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

Customizing Installation

The package installation can be customized in several ways. The package specification section describes how to define the necessary package metadata to configure the installation experience when the package is created. This section provides further detail on the two documents discussed in that section.

Product Name (install.inf)

The `install.inf` file is used to define the name of the product that is presented to the user during an interactive install (see install.conf below).

Ex.

install.inf

```
Example Product
```

If the `install.inf` is not defined the product name defaults to "QAD Software".

Installer and User Interface Settings (install.conf)

The `install.conf` defines settings that will be merged into the environment, ultimately getting defined in the following file when the package is installed.

```
[ENVIRONMENT]/build/config/system/environment.properties
```

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-"). In the following example `install.conf` file, the user is presented with a Yes/No question. Note the special "install-ui-end" setting to signal the end of the settings that will be presented to the user (and the @hide-all annotation to avoid showing that marker setting). The "install-command" setting, for example, configures the command that the installer will run to install the product (defaults to update) and the user will not be prompted for this setting. It is also important to point out that the user can execute the installer non-interactively with the option (-install-ui:false) in which case required settings must be supplied up front (i.e. -example.bi.integration:false) when the `install` command is submitted.

install.conf

```
# [QAD BI Integration]
#
# Enable integration with QAD BI?
# @option y:true:Yes
# @option n:false:No
# @optionstrict
# @default false
# @validation req
example.bi.integration=


# 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
...
```