Kobee

DevOps for z/OS mainframes using Kobee

How to prepare your Kobee installation for z/OS

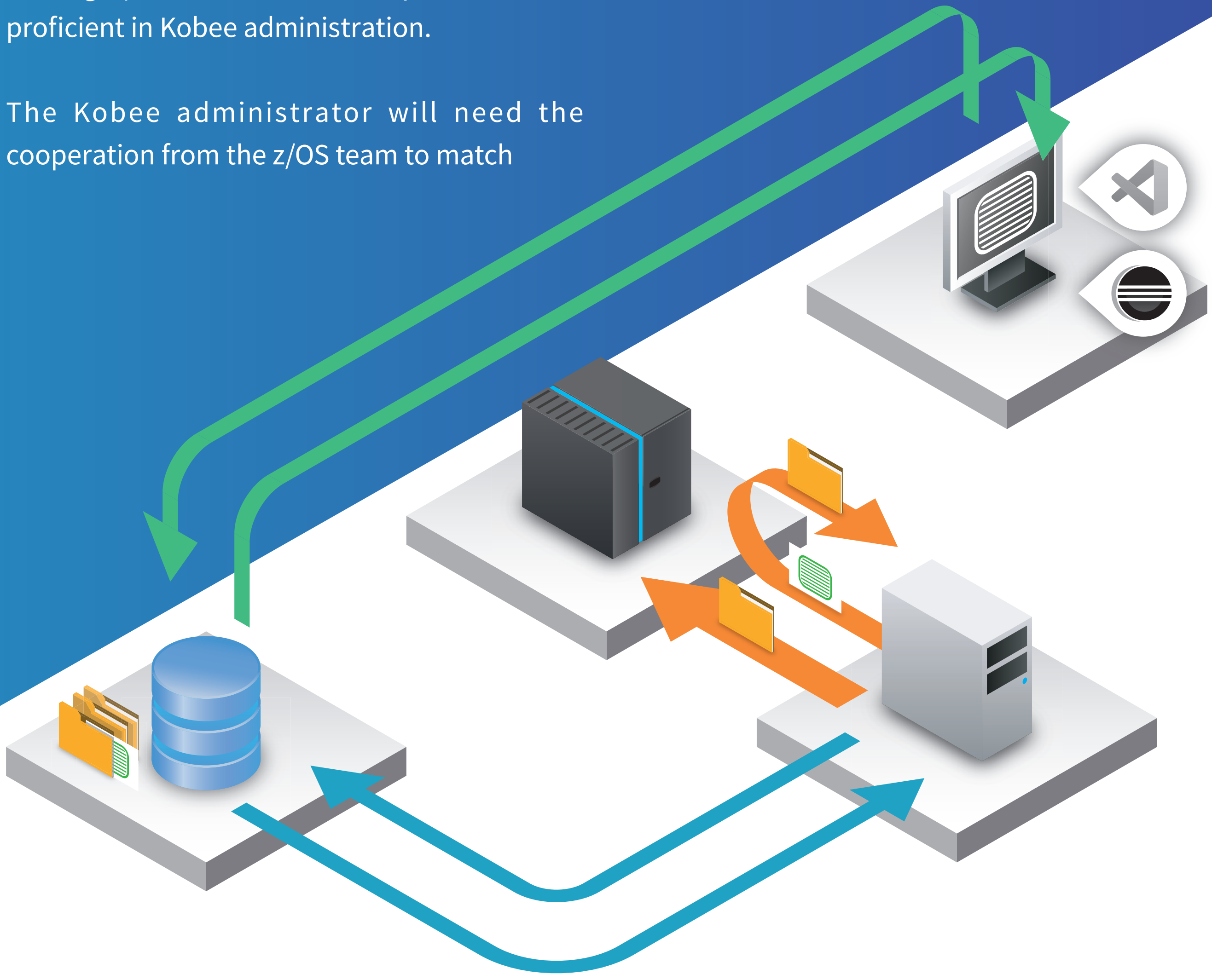
Table of contents

Summary

This document will show you how to set up a Kobee installation for z/OS, what the components are and how they work together.

Setting up this solution will require someone proficient in Kobee administration.

The Kobee administrator will need the cooperation from the z/OS team to match the current z/OS environment with Kobee. The following components should be available.



Key components

- **Kobee**
  The DevOps application set up with a connection to the Version Control Repository containing the z/OS sources. More info: https://docs.kobee.io/user-guide-en/6.0/GlobAdm_VCR.html

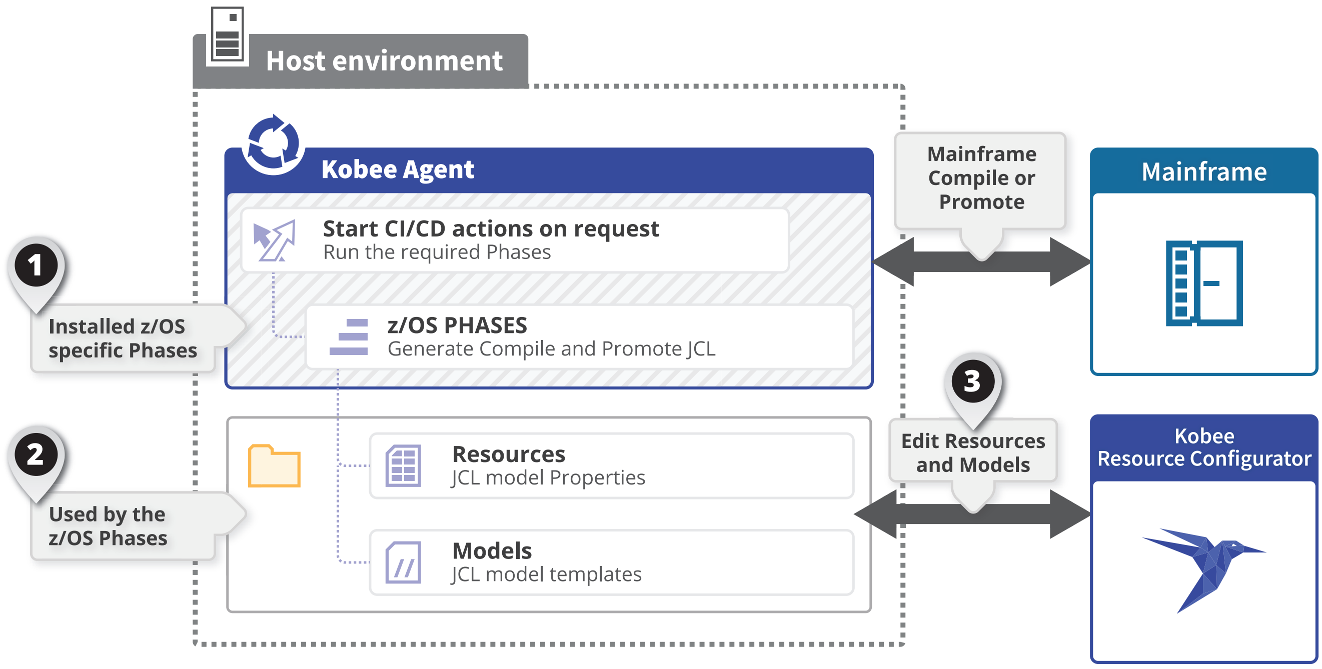
- **Version Control Repository**
  A Git or Subversion repository containing the z/OS source files.

- **Kobee solution Phases for z/OS**
  (See: image tag 1)
  Phases that are required to use Kobee for z/OS compile (build) and promote (deploy) actions. Phases are an integral part of the Kobee architecture, they represent the actions that need to be performed during a compile or promote request.

This unique concept allows for rapid workflow changes, dynamic behavior, fast traceability, and so on… More info: https://docs.kobee.io/how-to-use-and-develop-phases-en/5.7/UseDevelopPhases.html

- **Kobee Resource Configurator**
  (See: image tags 2 & 3)
  An application to edit and generate the required models and resources used by the z/OS solution Phases.

- **Phase Scripts**
  Although not strictly necessary, it is set up by default. The folder contains scripts which are copies of the scripts that come bundled with the Phases. This allows script customization that will affect all Phases at once, if needed. Phase scripts will not be covered in this document.



Host environment
Kobee Agent
Start CI/CD actions on request
Run the required Phases
z/OS PHASES
Generate Compile and Promote JCL
Resources
JCL model Properties
Models
JCL model templates
1
Installed z/OS specific Phases
2
Used by the z/OS Phases
3
Edit Resources and Models
Mainframe Compile or Promote
Mainframe
Kobee Resource Configurator

Other components

Ant tools

The z/OS Phases make use of libraries which are not part of the standard Ant distribution. To verify the presence of these files, in Kobee go to the top menu, click the “Global Administration”. Next click “Overview” in the “Scripting Tools” panel.

On the “Scripting Tools Overview” page, click the “Edit” icon in from of the Ant tool. On the edit page check if the “Lib Path” contains a system location value.

The location (on your file system) should contain the following files:

- ant-ikan-tools.jar
- jxl-2.6.10.jar
- ant-contrib-1.0b3.jar

When missing, look for these files in the Kobee distribution archive.

<u>**NOTE:**</u> Note that in this documentation the assumption is that the Kobee Agent and the Kobee Server are running on the same machine. Optionally you can run the Agent on a different machine or even run the agent on z/OS using UNIX System Services (USS).

Setting up the z/OS solution

The z/OS solution Phases

First the z/OS solution Phases need to be imported (from the distribution archive) into Kobee. Since this documentation only focusses on these specific Phases they will be referred to as "Phases" for the remainder of this document. Only a few Phases are needed to make a minimal working z/OS DevOps solution, these are mandatory and always need to be installed. In a real scenario however you would want to install other optional Phases to match your z/OS environment.

The required Phases are:

For the compile actions:
- Copy from Source to Target
- Copy Sources to PDS for compilation
- Maps and Programs compilation

For the promote actions:
- Copy from Source to Target
- Promotion of components and load-modules

For a complete list of Phases including details, visit:
https://docs.kobee.io/integration-zos-phases-catalog/2.1/zOS_Phases_Intro.html

To see how you can import Phases in Kobee, visit:
https://docs.kobee.io/user-guide-en/6.0/GlobAdm_Phases.html#_globadm_phases_importing

Creating a project in Kobee

Once imported, a new project has to be created; this is done in the Global Administration section of Kobee.
For detailed information, visit:
https://docs.kobee.io/user-guide-en/6.0/GlobAdm_Project.html

Instead of creating a new project it is possible to import the "DemoZOS" project from the Kobee distribution archive, follow the related steps in document (https://www.kobee.io/documents/integrations/zos/Windows_distribution_Kobee_ZOS2.2.0.pdf) and then return to this document when finished and move to the next chapter.

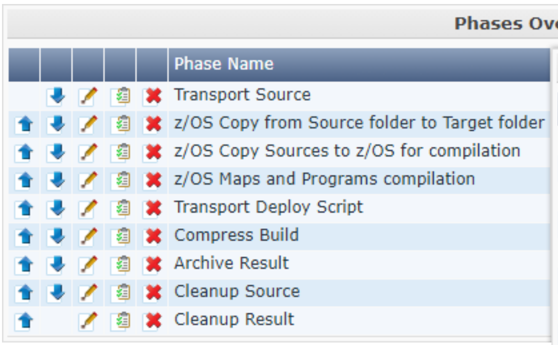
It's important to set the project type to "Package-based". During the project creation process, Kobee automatically sets up a Project Stream with a base Lifecycle for this new project. In this empty base lifecycle, levels (Build, Test, and Production) and environments (Build and Deploy) need to be created manually. The project name cannot contain empty spaces because it is also used by the solution for folders, Phases and properties.

We highly recommend reading the Project Administration chapter, starting at the Project Streams before moving on.
https://docs.kobee.io/user-guide-en/6.0/ProjAdm_ProjMgt_ProjectStream.html

Adding Phase to the Project Environments

The Phases need to be added to the appropriate (Build and Deploy) environments. Since Phases may be dependable on other Phases it is important that they are arranged in the correct order.

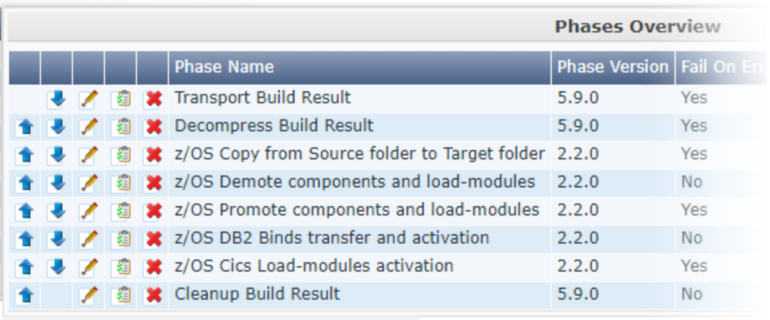
Below is an example of what the order could look like, this example is based on the setup that IKAN uses to demo its z/OS solution on Microsoft Azure.



Phases Overview

Phase Name
Transport Source
z/OS Copy from Source folder to Target folder
z/OS Copy Sources to z/OS for compilation
z/OS Maps and Programs compilation
Transport Deploy Script
Compress Build
Archive Result
Cleanup Source
Cleanup Result

Phases on a Build Environment



Phases Overview

Phase Name	Phase Version	Fail On Err
Transport Build Result	5.9.0	Yes
Decompress Build Result	5.9.0	Yes
z/OS Copy from Source folder to Target folder	2.2.0	Yes
z/OS Demote components and load-modules	2.2.0	No
z/OS Promote components and load-modules	2.2.0	Yes
z/OS DB2 Binds transfer and activation	2.2.0	No
z/OS Cics Load-modules activation	2.2.0	Yes
Cleanup Build Result	5.9.0	No

Phases on a Deploy Environment

In the Project Administration scope, go to: “Build (or Deploy) environments” > “Overview” > “Edit Phases”. On “Build Environment Phases Overview”, click the “Insert Phase” link.

The Build Phases should chronologically be in between the following core Phases:
- Transport Source
- Compress Build

Also, the following core Phases must be deleted for the Build Environment:
- Verify Build Script
- Execute Script
- Transport Deploy Script

The Deploy Phases should chronologically be in between the following core Phases:
- Decompress Build Result
- Cleanup Build Result

Also, the following core Phases must be deleted for the Deploy Environment:
- Verify Deploy Script
- Execute Script

Models and Resources used by the z/OS solution Phases

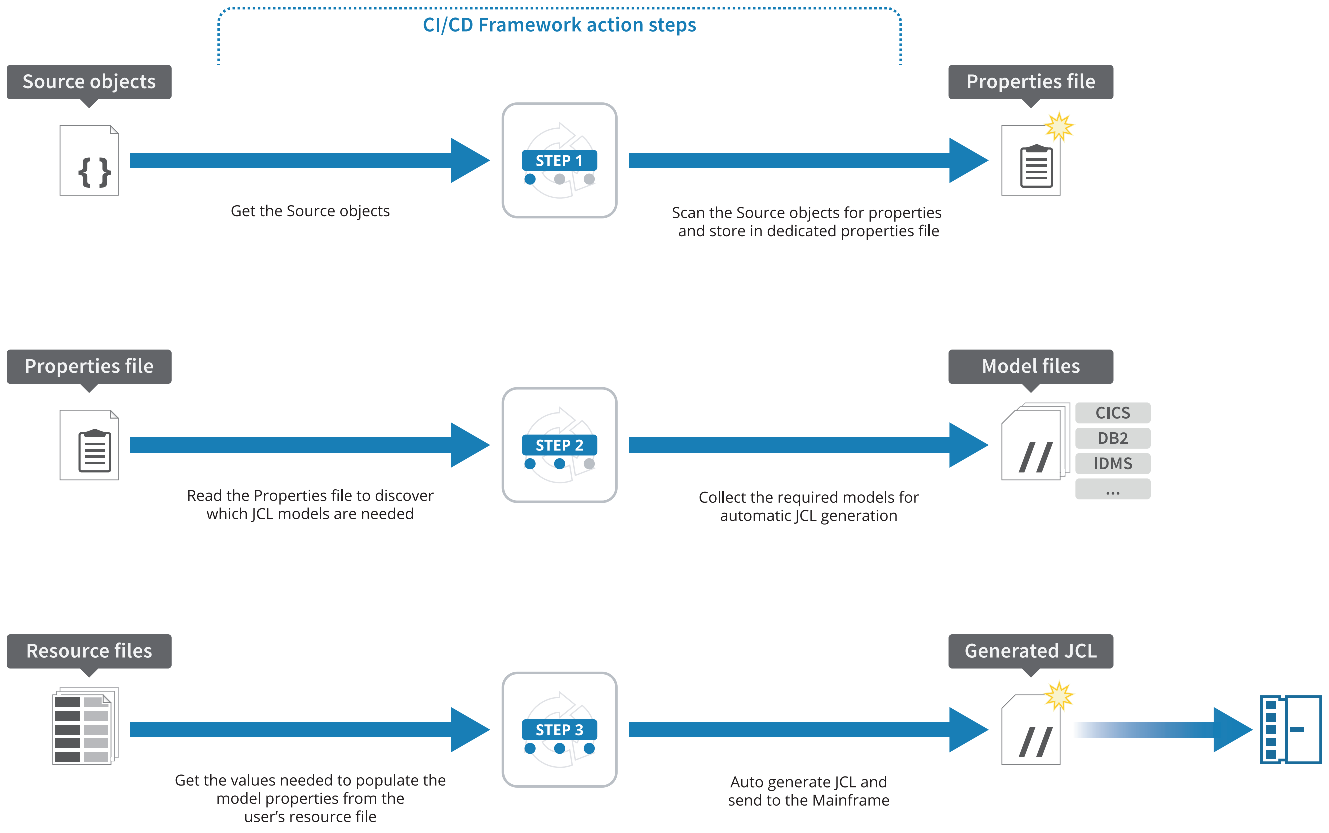
Broadly speaking the main purpose of the Phases is to generate JCL code which is then transferred to the z/OS mainframe where it will start the compile or promote process.

The compile or promote JCL code is generated based on the properties defined for your source code. These properties are either set by Kobee (z/OS Phases) by analyzing your source code or picked up by Kobee in case the user provides a properties file for each source. Through these properties Kobee (z/OS Phases) will define which JCL steps are needed for the specific source code.

For every JCL step that is needed, a JCL Model is retrieved from the JCL Models archive and then added to the to-be-generated JCL code.A JCL Model is basically a JCL card template with property placeholders. The placeholders are used to customize the JCL card according to the type of source code and the z/OS environment it will run in.

The properties needed for this customization are retrieved from the Resources archive (and from property files which are generated per program source by Kobee). The entries in the Resources archive need to match your z/OS environment of course.

The image below shows the process simplified.


CI/CD Framework action steps
Source objects
Get the Source objects
STEP 1
Scan the Source objects for properties and store in dedicated properties file
Properties file
Properties file
Read the Properties file to discover which JCL models are needed
STEP 2
Collect the required models for automatic JCL generation
Model files
CICS
DB2
IDMS
...
Resource files
Get the values needed to populate the model properties from the user's resource file
STEP 3
Auto generate JCL and send to the Mainframe
Generated JCL

Kobee for z/OS provides many JCL Models that can be adapted to your z/OS environment and it also possible to add your own JCL Models.

While JCL models are recognizable by a "_jcl.model" suffix. There are also other models used by the Phases which support different functions, again the suffix of these models reflects their functionality e.g., a sysin model will have a "_sysin" suffix, an FTP model will have a "_ftp" suffix and so on...

How to get started

Kobee Resource Configurator (KRC), comes with a default Configuration Set. When there is an existing installation of the Kobee for z/OS solution prior to the release of KRC, it is best to create a new Configuration Set as it will offer a blank set to which the existing files can be imported into. Importing files will be addresses in the "Creating backups and performing imports in KRC" chapter.

In other cases it is a good practice to copy this default Configuration Set to a new set. The original set can then be used for future references when needed. Using a copy is a quicker way to become operational as it will only require customization rather that starting from scratch.

List the JCL's you are currently using for compilation and promotion. Once this list is available check for matching JCL Models in Kobee Resource Configurator. Then make the necessary modifications to make sure that Kobee for z/OS will act similar to your current JCL's and aligns with your standards.

Models adhere a naming convention: "stepnameWHAT_jcl.model". For example, a pre-compile step for COBOL would translate to: "precompileCOBOL_jcl.model"

As mentioned before in case you can't find a specific model, a new model should be created for the missing JCL step. A good practice is to find a similar model and to look for the required JCL properties, which are needed by the Resource configuration.

You could start with a compile JCL. For the implementation, you will need all steps (remember there is a JCL model for each step) and the SYSINs; the job card and job output can be omitted as they are generated automatically. JCL steps you would need to identify are: compile, endJcl, expand, copyLct, debugger, linkEdit, listing, preCompile, preCompileCics, preCompileDb, etc.

When creating a new model, its name should preferably correspond to the type of step, for example if it's a compile model then the name should start with 'compile'. It is a good practice and it will make linking a model to a step much more obvious later on when you start configuring the solution.

For each JCL step, identify the exec card parameters like: program used for compilation, the DD statements for the file/member datasets and the parameters like: PARM, RC... parameters such as STEPLIB, SYSIN… and SYSLIB content used for compiles and binds.

Determine which SYSLIB entries are needed. Generally there should be an entry with a default temporary and empty file definition (DSN) of the SYSLIB card, followed by one or a combination of the built-in SYSLIB generation model parameters below. For an example you can take a look at the compile models for Cobol or Assembler.

These are the available parameters you can use:

- "${copylibs}"
  Generates the SYSLIB code for the library that contains the copybooks that are required by the model.

- "${linklibs}"
  Generates the SYSLIB code for the system library that contains the executable modules for z/OS components and utilities that are required by the model. Sometimes you may encounter a "${incllibs}" entry in the models, this is an older parameter that is now equal to the "${linklibs}" parameter.

The SYSLIB code generated by the model parameters above can be extended when extra features are needed.

- "${extendedCopylibs}" (derived from: "extendedCopylibs_jcl.model")
  Generates the SYSLIB code for extra program features (CICS, DB2,...) that are required by the model and which needs to be added after the code generated by ${copylibs}.

- "${extendedLinklibs}" (derived from: "extendedLinklibs_jcl.model")
  Generates the SYSLIB code for extra system features (CICS, DB2,...) that are required by the model and which needs to be added after the code generated by ${linklibs}.

Code generated by both the basic and extended SYSLIB parameters relies on information configured in the "environment_build_zos.properties" file. The two extended SYSLIB model parameters additionally retrieve data from the specific feature Build model files (e.g. the CICS feature will use the "environment_build_cics" properties file).

Example of generated SYSLIB entries (in the compile Cobol model):

```
//SYSLIB    DD DISP=(,PASS),DSN=&&SYSLIBCP,
//              UNIT=SYSALLDA,SPACE=(TRK,(1,1,1),RLSE),
//              DCB=(DSORG=PO,RECFM=FB,LRECL=80,BLKSIZE=27920)

//          DD  DISP=SHR,DSN=IKANALM.DEMOS.P0000046.COPYLIB
//          DD  DISP=SHR,DSN=IKANALM.DEMOS.TEST.COPYLIB
//          DD  DISP=SHR,DSN=IKANALM.DEMOS.P0000046.COPYLIB
//          DD  DISP=SHR,DSN=IKANALM.DEMOS.TEST.COPYLIB
//          DD  DISP=SHR,DSN=IKANALM.DEMOS.P0000046.DCLGEN
//          DD  DISP=SHR,DSN=IKANALM.DEMOS.TEST.DCLGEN

//*         DD  DISP=SHR,DSN=SYS1.COB2LIB
//*         DD DISP=SHR,DSN=DFH320.CICS.*
//**        DD DISP=SHR,DSN=DSN810.DCLGEN
//*         DD DISP=SHR,DSN=CAI.DATACOM.COPY
//**        DD DISP=SHR,DSN=IKANALM.DEMOS.PROD.MACRO
//*         DD DISP=SHR,DSN=IKANALM.DEMOS.PROD.COPYCOB
//**        DD DISP=SHR,DSN=IKANALM.DEMOS.PROD.COPYPLI
//          DD DISP=SHR,DSN=SYSV.CPW.ECC300.MLCX870.COPYLIB
```

Content from model
(optional with inline parameters)

Generated by **${copylibs}** parameter

Generated by **${extendedCopylibs}** parameter

During the compilation phase, all parameters in a model will be replaced by generated values based on the program/map source parameters, object types, environments and language.

You don’t need the job card, print listings of the Pre-compile, Compile and Bind.

Other information you should investigate:

- **Languages to compile**
  Kobee for z/OS supports a number of languages like ASM, COBOL, PL/1... by default.
  Check if the language you want to compile is available or if an available language needs changes.

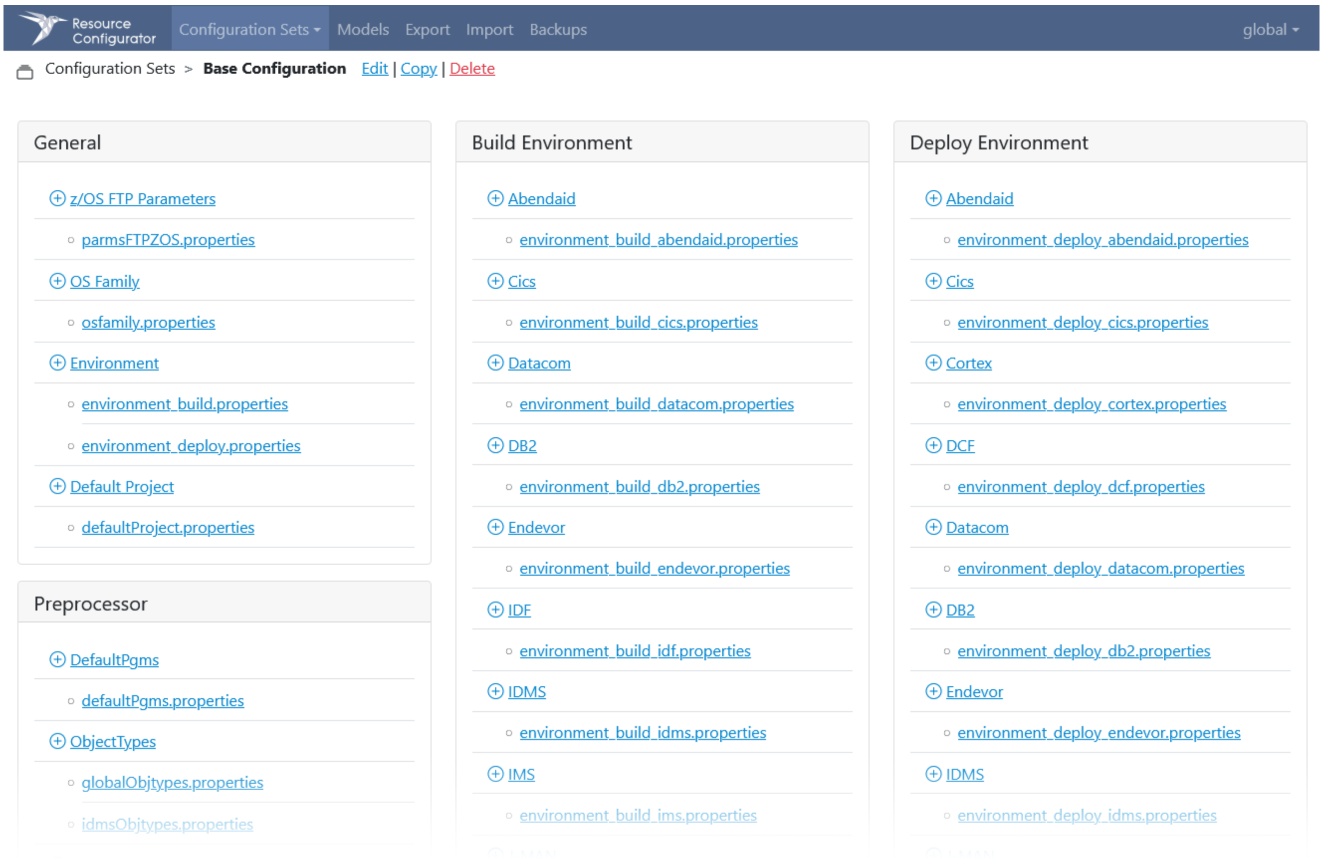
- **Compile and Bind steps**
  The same principles (see above) apply.

- **Extra tools or features**
  Check if other tools are required like a debugger for example.

- **Specific files**
  It might be possible that next to the compile result (load and listing) you also need other files for deployment on z/OS. In this case you need to make sure that they are part of the build result. Note: this build result is stored in the Build archive location on the Kobee Server.

Verifying the JCL Models

The Resources and Models are managed by the Kobee Resource Configurator. Open the Kobee Resource Configurator (KRC) and log in.



General

- z/OS FTP Parameters
  - parmsFTPZOS.properties
- OS Family
  - osfamily.properties
- Environment
  - environment_build.properties
  - environment_deploy.properties
- Default Project
  - defaultProject.properties

Preprocessor

- DefaultPgms
  - defaultPgms.properties
- ObjectTypes
  - globalObjtypes.properties
  - idmsObjtypes.properties

Build Environment

- Abendaid
  - environment_build_abendaid.properties
- Cics
  - environment_build_cics.properties
- Datacom
  - environment_build_datacom.properties
- DB2
  - environment_build_db2.properties
- Endevor
  - environment_build_endevor.properties
- IDF
  - environment_build_idf.properties
- IDMS
  - environment_build_idms.properties
- IMS
  - environment_build_ims.properties

Deploy Environment

- Abendaid
  - environment_deploy_abendaid.properties
- Cics
  - environment_deploy_cics.properties
- Cortex
  - environment_deploy_cortex.properties
- DCF
  - environment_deploy_dcf.properties
- Datacom
  - environment_deploy_datacom.properties
- DB2
  - environment_deploy_db2.properties
- Endevor
  - environment_deploy_endevor.properties
- IDMS
  - environment_deploy_idms.properties

Select your Configuration Set and then click on “Models” in the top menu, this will display the available JCL Models. Pick a random model to view its contents, for example: “compileAsm_jcl.model”.

Using the top panel “Model Info” actions it’s possible to copy, download, or upload the selected model. Note that it is not possible to edit the JCL Model contents using KRC, you will need to download and upload the model file to do so. The edit action in the “Model Info” panel can be used to change the model's name and to set the export path (within the export folder hierarchy) which is by default: “BUILD”.

In the model content you will see the property placeholders mentioned earlier, for example: “${pgm.language}”, these may not be changed or removed.

Entries starting with “RCMAX=” or "WARNING=" are used by Kobee internally, they should be left as is.

Go through all the JCL Models in the list to verify if they are compliant to your setup.
There are models available for the Build (Compile) and Deploy (Promote) steps, the distinction between the two is noticeable in the export path. We recommend using this convention when creating new models. Please be aware when changing or creating a JCL model that step names used in the JCL code are limited to 8 characters.

There are additional subfolders for specific cases:
All models related to communication with the mainframe (FTP) are grouped in the “ftp”, “mfdas”, “zosmf” and “uss” folders.

Models used by the Phases for double compilation scenarios are located in the “double” folder.
See the appendix for a practical example on how properties are used in the models.

Configuring the Resources

As explained earlier Kobee Resource Configurator (KRC), comes with a default configuration set. It is a good practice to copy this default configuration to a new set and to customize this new set for your environment. The original set can then be used for future references when needed.

Configuring the Resources is a necessity; Kobee for z/OS will not work in your environment using the default configuration.

Click on “Configuration Sets” and select the “Base Configuration”. Click the “Copy” link below the top menu and choose a name for the new set. When finished, KRC will display your new set in the breadcrumb; always make sure that you’re making changes to the right set.

It’s important to mention that when on a page, the toggle switches on the right side of the values should be verified since they represent the status of a value (active/inactive). if you want to use an inactive item you need to make sure that the toggle switch is set to active mode before saving the property, otherwise the parameter will not be exported to the property file nor will the parameter value you have entered be saved.

The Resources page is divided into five categories, and each of the categories contains default types with their entries as provided with the default Configuration Set.

1. General

- **z/OS FTP Parameters**
  - `parmsFTPZOS.properties`
    Compile and Promote JCL is sent to the z/OS mainframe using an FTP connection. During this process the JCL gets executed and components are sent and received. This property file represents the connection details for one z/OS LPAR. For multiple LPAR's additional property files will need to be created or the user and LPAR IP should be configured as an environment parameter.

- **OS Family**
  - `osfamily.properties`
    Details per host operating system (of the Kobee Agent) used for the generation of the FTP connection script (shell/console). For example, the file extension.

- **Environment**
  - `environment_build.properties`
    Properties to configure the z/OS build environment and its active features (CICS, DB2...). To set the properties of a specific feature instead, you have to use the 'environment_build_feature.properties' (e.g., 'environment_build_cics.properties') in the Build Environment panel.

    To use a specific feature it needs to be activated by setting the value to ‘True’, for example by setting the option ‘True‘ for the ‘Cics Active’ feature in case you need it.

    When a feature is enabled the corresponding properties file (“environment_build_[featurename].properties”) will be loaded. This file can be found in the "Build Environment" category.

The compilerTypes property will list the available values for the compilers on the z/OS build environment. If the generated (using the defaultPgms compilerType) or imported property file for a source program has a different compilerType then the compile process will not work.

› `environment_deploy.properties`
Properties to configure the z/OS deploy environment and its active features (CICS, DB2...). To set the properties of a specific feature instead, you have to use the 'environment_deploy_feature.properties' (e.g., 'environment_deploy_cics.properties') in the Deploy Environment panel.

When a feature is enabled the corresponding properties file (“environment_deploy_[featurename].properties”) will be loaded. This file can be found in the "Deploy Environment" category.

- **Default Project**

› `defaultProject.properties`
Properties for extending JCL properties (Project/Package/Release/Version), setting z/OS properties and changing default language properties.

These global properties are always available for the Phases, models and external Phase scripts. Many of these properties are derived from built-in properties in Kobee, unless they are changed in this Default Project properties file.

For a full list, see the appendix.

Although they are used internally the properties can also be used in script, models, etc. if needed, by using their export name. The export name can be found by clicking the “Show Descriptions” link button and searching for the name below the input field. The name however can only be used when surrounded by round braces and a dollar sign (e.g., projectVersion should become: ${projectVersion}).

2. Preprocessor

- **DefaultPgms**

› `defaultPgms.properties`
The default z/OS program properties (logical language, object type, etc...) and features that will be used if the automatic detection fails during code analysis (by using the ‘sourceRules’ rules for detection) and if there is no property file found in the source code package (in Kobee). When using property files be aware that they should have a matching file name (e.g. "cobolfile.cblbatch" and "cobolfile.properties").

Note that for complex cases the detection process can be customized in the "getZosProgramProperties.xml" script. The automatic detection process or using property files are preferred rather than using DefaultPgms.

The defaultPgms property file contains “pgms.compilation”, “pgms.linkedit” and “pgms.execution” properties to determine if the component to build will have a ‘compilation’ step, a ‘linkedit’ step and for the JCL an ‘execution’ process. If a source code package has a properties file, the properties will have a "pgm.xxx" prefix instead.

The Object Type must relate to an existing Object Type in the Global Object Types properties file. The relation happens through the (luw=suffix) property.

Then the “pgm.compilerType” sets the compiler type but first verifies if this type is defined in the "environment_build.properties" file. By default, the "environment_deploy.properties" file does not have a compiler type.

- **ObjectTypes**
  - `globalObjtypes.properties`
    The Global Object Types file is used as a library by the z/OS solution Phases in which entries are defined by function type (SRC: “Source program”, JCL: “JCL’s”, LIST: “Listings”...) and are associated with a unique file suffix (or extension) each. Optionally Object Types can be matched to other Object Types, for example a Source program can be matched with a Load-module and a DB2 DBRM, etc.

    This file contains more than one possible definition identifying a type of source or component to manage. We advise to delete logical object types that are not not needed. You can have different object types files if necessary.

    In case of a double compilation the Object Type must match with two Load and two List objects.Read the Appendix for a full description of all the function types and properties.

    The Phases (actually the first Phase that runs) will select the current Object Type from this file once the logical language has been found after the source file analysis. Because the selection process will use the suffix value from the "OBJECT.luw" entry as the identifier, this value needs to be unique in the Global Object Types library. The only scenario where the same suffix could be tied to multiple object types is when a "Source Language Options" (SRC_LANG) child type is used to extend a "Source programs" (SRC) object type, for example when a single PDS has multiple source languages.

    In the last case a main Object Type with the SRC function is thus completed with a child Object Type containing a logical language name (defined in ‘languageZOS.properties’) in its property name, for example: ‘SRB_ASSEMBLE’. Batch and CICS sources can have a separated targeted PDS for their load-modules, in this case the Object types must be managed separately like SRB and SRC object types.

  - `idmsObjtypes.properties`
    Defines the Object Types (in the same manner as ‘globalObjtypes.properties’) but for IDMS specifically.

    Adding custom properties for a Global Object type
    It’s possible to add custom properties to a property type of an object type by going to the bottom of the page and clicking “Add a Custom Property”.

    “Name” should have the following syntax “propertytype.propertyname”, where “propertytype” refers to the group to which the property belongs to. For example if we want to add a custom property to the “luw” property type, the name would look like “luw.propertyname”.

    To use this custom property, in a model or elsewhere, we will need to add the object type explicitly to the placeholder, prepended with the string “objtype.”. For instance, if the object type is Cobol the placeholder will have to look like: ${objtype.COBOL.luw.propertyname}

- **Source Rules**
  - `sourceRules.properties`
    Ant (regex) rules for analyzing the Sources to determine a logical language and its type (program or map) and other options. When the rule is found, the logical language is set as property ‘pgm.language=LANGUAGE’ to be used by the Phases, except if there is a predefined properties file included in the source package. We advise to have same logical languages into this file and the "languagesZOS.properties" file.

It’s possible to add rules to the sourceRules when a logical language is not supported or when the default rules are insufficient.

A logical language is identified using four grouped subproperties:

[property].copybook
- For identifying copybook lines in the source file. This is optional.

[property].lang:
- Language name (must relate to a logical language in the languagesZOS property file)

[key].pattern
- Java regex rules for searching inside the source file. A rule can be ignored by using something like .*¨-¨-¨-¨-¨.*.
  Then the script must resolve the logical language.

[key].type
- Type of source (program, map, dds(as400)).

If all else fails it is possible to make code changes to the “getZosProgramProperties.xml” Ant script directly. This file can be found in the phase scripts folder (by default named: ‘phaseScripts’).

Note that when adding new rules there also must be a corresponding Object Type (in ‘globalObjtypes.properties’) and a Logical language entry (in: ‘languagesZOS.properties’).

- **DB2 Bind Rules**
  › `db2Rules.properties`
  Properties and rules for managing Bind information from the program source.

- **J-MAN Rules**
  › `jmanRules.properties`
  Properties and rules for managing Jman models information from the Jman source.

3. Compilation Configuration

- **Languages**
  › `languagesZOS.properties`
  Definitions of the JCL steps that are available per logical languages. The Phases use these steps, depending on the required features after the source program analysis, to chronologically build the compile/promote JCL.

  It is not possible to add a new step type to a language in KRC, the available steps types should be sufficient. However, if this is really necessary you will need to contact IKAN support.

  There is no step required for the Job Card; it is added automatically.

  Each step is configured using properties. Some of these properties are mandatory, see the table.

Property	Description
JCL Step	Key word that defines the JCL step behavior*
Step Sequence	The order of this step in the JCL sequence*
JCL Model File	JCL Model(s) to use for building the JCL step*
Main Program Step	z/OS program to run on the mainframe during the JCL step
Parameters	Options

JCL Step:
Supported steps types are:
copySource, expand, compile, preCompile, preCompileCics, preCompileDb, debugger, linkEdit, listing, copyLct, endJcl or null.

The null value is useful for managing step properties without adding a model to the generated JCL.

Step Sequence:
Reserved sequence numbers are: 10 for the “copy” step and 15 for the “expandSource” step.

Except when there is a “service” step with sequence number 25, then the “copy” step sequence number has to be 26, or a “datacom” step with sequence number 20, then the “expandSource” step sequence number has to be 21.

In case of multiple models, the numseq parameter is incremented automatically by 1 for each model.

By default when sequence numbers are implemented they are set to increment by 5 for each additional step, this allows room for 4 models in a single step, which should be sufficient.

If a double compilation (batch and transactional or double CICS) is necessary, the numseq parameter can have 2 numbers (for example: 50,80). However, in order to use the double compilation feature the option needs to be enabled (set to “true”) in the defaultPgms or predefined (with the source) properties file.

JCL Model File:
The beginning of a model's name (e.g., ‘compileCobol_jcl.model’) will most likely correspond to the type of step (using the example above:‘compile’). This is by default in the base configuration set.

By default there are specific models for each logical language, especially in case of the pre-compile, compile, bind and debug steps.

Multiple models are allowed per step.

In case a double compilation the compilation phase will use an additional model, located in the “double” sub-folder. For example when a “compileCobol_jcl” model has been defined, an additional “compileCobol_jcl_2” will be added automatically.

Optional properties:
The rcmax parameter may be optional yet it is crucial as it is used to set and later compare the return code for the COND parameter (whether the step should be skipped based on the previous step) in the generated JCL.

It is used by Kobee for generating a list of return codes per step that is used in comparison with the return code values in the JES logs coming from the mainframe. It is set using a "MAXRC=..." string in the EXEC card, the same applies to the warning parameter: "WARNING=...".

Properties can be used in the models through the use of placeholders. For example the placeholder “${compilation.lang.program}” in a model refers to the “program” property in the “compilation” step of a language. The “lang” part in the placeholder must be left as is, it is internally set equal to the detected source language.

For a description of the other optional properties use the “Show Descriptions” option in any language configuration page.

Adding custom properties
It’s possible to add custom properties to a language step by going to the bottom of the page and clicking “Add a Custom Property”.

“Name” should be have the following syntax “groupname.propertyname”, where “groupname” refers to the group to which the property belongs to. For example if we want to add a custom property to the compilation step group, the name would look like “compilation.propertyname”.

Adding a new logical language
It is possible to create a new logical language but for the system to detect this new language and its properties it is also necessary to create new rules. To create new rules a new Source Rules configuration object has to be created. If this seems to be inadequate it is also possible to write a custom solution in the “getZosProgramProperties.xml” file which is located in the phase scripts folder.

› `languagesZOSJOB.properties`
Definitions of the JCL steps that are available for a JCL source logical language. Apart from this, the same principle as with the 'languagesZOS.properties' also applies to this property file.

4. Build Environment Configuration

- ***[FEATURE]***
  › `environment_build_[featurename].properties`
  To set the properties of a specific feature (abendaid, for example) on the z/OS build environment. Note that the feature also needs to be activated in the 'environment_build.properties' file, however for mandatory z/OS features this is not necessary.

- **z/OS**
  › `environment_build_zos.properties`
  Management of dataset info, for example to specify your PDS(E) naming conventions, z/OS route id and volume, Load-modules transfer prefix, path levels used for SYSLIB copybooks, etc.

When JCL is generated any DSN specified here is automatically appended with the Last Level Qualifier (LLQ) of the corresponding object type and the suffix defined in the 'globalObjtypes.properties'. In the case of USS (Unix System Services) the path and suffix properties are used.

5. Deploy Environment Configuration

- ***[FEATURE]***
  - `environment_deploy_[featurename].properties`
    To set the properties of a specific feature (abendaid, for example) on the z/OS deploy environment. Note that the feature also needs to be activated in the 'environment_deploy.properties' file, however for mandatory z/OS features this is not necessary.

- **z/OS**
  - `environment_deploy_zos.properties`
    The same principle as with the 'environment_build_zos.properties' also applies to this property file.

Exporting the Configuration Set

Once the Resources are configured and all Models are verified, the configuration set must be exported to a location on the Kobee Agent. It is a good practice to use a common folder for all Kobee (z/OS) related files.

In Kobee Resource Configurator (KRC), click on “ Export” in the top menu, this will display the defined export locations. Click the “Export this Configuration Set” button, choose the desired location and click the “Export” button.

As you may have noticed you can choose to deselect the configuration files and the models if needed. The exported files will be divided into two folders, you can change the folder names if desired by clicking “Edit” in the breadcrumb on the “Configuration Sets” page of the configuration set.

When finished successfully you will need to configure Kobee to use this configuration set, this has to be done by adding parameters on various layers (global, environment and Phase layers) in Kobee.

Phase parameters in Kobee

Kobee Phases are predefined, parameter-driven, reusable building blocks. The ability to use parameters to control the behavior of a Phase is a powerful feature.

Parameters can be defined on a machine, on an environment and on a Phase. When defined at the machine layer the parameter will be used everywhere, except when it is overwritten by an environment, and the environment parameter will be used except when it is overwritten by a Phase.

For a full explanation see:
https://docs.kobee.io/how-to-phase-parameters/1.0/PhaseParameters.html

For the z/OS solution, parameters will need to be set on all layers.

Location parameters

Locations or z/OS property files can be set when on all levels but normally they will be set globally (machine or environment). Below is a list of common location that can be set, some of them will return in the practical implementation examples in the next paragraphs. The "propsfile.xxx" and "pty.xxx" files are set by default using the "dir.zosResources" location in their path.

Property	Description
ikanalm.home	Kobee Server or Agent
dir.zosModels	Models (this location is mandatory)
dir.zosResources	Resources (this location is mandatory)
dir.phaseScripts	Phase Scripts
propsfile.parmsFTP	FTP connection file (parmsFTPZOS.properties) (*1)
propsfile.objtypes	Global Object Types file (globalObjtypes.properties)
propsfile.rules	Source Rules file (sourceRules.properties)
propsfile.defaultPgm	Source property file (defaultPgms.properties)
propsfile.db2Rules	DB2 rules file (db2Rules.properties) (= optional)
propsfile.jmanRules	JMAN rules file (jmanRules.properties) (= optional)
pty.defaultProject	Default Project property file (defaultProject.properties)
propsfile.languages	Languages property file (languagesZOS.properties) (for the Build level)

(*1): When using multiple LPAR's for different environments it's best to create multiple FTP parameter files and set them on the corresponding envinronments in Kobee.

Machine Parameters

In the top menu, click the “Global Administration”. Next click “Overview” in the “Machines”, then panel, then click on the “View Parameters” icon for the agent (or server/agent if combined) machine. In the “Machine Parameters Overview” table click the “Create Parameter” icon to add the following parameters:

Key	Value	Mandatory	Editable	Dynamic	Description
ikanalm.home	C:/kobee	Y			*1
dir.phaseScripts	C:/ikan/ALM_system/PhaseScripts	Y			*2

(*1): Sets the home directory of the Kobee installation folder. This is used by Ant.

(*2) : Use external instead of the internal Phase scripts. This allows the modification of the “getZosProgramProperties.xml" file when the Source Rules are inadequate to define the logical language.

Build Environment Parameters

Assuming that the z/OS project is already set up, go to the project using the top menu “Project Administration”, and click the “Edit” icon. On the “Project Info” page, click on “Build Environments > Build Parameters”. Add the following parameters:

Key	Value	M	E	D	?
subjcl.parallel	1	Y			*1
project.zos.package.template	P????	Y			
stopOnErrorCompile	true;false	Y		Y	*2
dir.zosModels	C:/ikan/ALM_system/ZOS/PhaseModels/BUILD	Y			*3
dir.zosResources	C:/ikan/ALM_system/ZOS/PhaseResources	Y			*4
includedFiles	**/*.*		Y		
script.getZosProgramProperties	${dir.phaseScripts}/getZosProgramProperties.xml	Y			*5
dir.jeslogs	C:/ikan/ALM_system/jeslogs/${alm.project.name}/${alm.levelRequest.levelName}	Y			*6
ftp.zosconvcmd	quote SITE SBD=(IBM-1047,ISO8859-1)		Y		*7
usePreviousArchive	false;true			Y	
propsfile.environment	${dir.zosResources}/BUILD/environment_build.properties	Y			*8
propsfile.objtypes	${dir.zosResources}/globalObjtypes.properties	Y			*9
propsfile.languages	${dir.zosResources}/BUILD/languagesZOS.properties	Y			*10
ftp.active	true;false	Y		Y	*11
save.jeslogs	TRUE	Y			*12

(*1): Number of compiles that can run in parallel

(*2): Option to stop the compilation phase when a compile error occurs

(*3): Phase Models location

(*4): Resource files location

(*5): getZosProgramProperties.xml location

(*6): JES logs location. The values of the “dir.jeslogs”: ”${alm.project.name}” and “${alm.levelRequest.levelName}” are parameters available in Kobee. For a full list of parameters see: https://docs.kobee.io/user-guide-en/6.0/App_PredefLevelParams.html

(*7): FTP subcommand (zOS encoding, transfer encoding) for mainframe

(*8): Environment build properties location

(*9): Global Object Types location

(*10): languagesZOS properties file location

(*11): Option to disable the FTP transfer of JCL (useful for troubleshooting)

(*12): Save the JES logs

NOTE: Read the “Project Administration” chapter of the User Guide on docs.kobee.io for help on setting up a project.

Test and Production Environment Parameters

The same way of working as with the Build Environment Parameters applies to these parameters.

Key	Value	M	E	D	?
project.zos.package.template	P????	Y			
dir.zosModels	C:/ikan/ALM_system/ZOS/PhaseModels/DEPLOY	Y			*1
dir.zosResources	C:/ikan/ALM_system/ZOS/PhaseResources	Y			*2
dir.jeslogs	C:/ikan/ALM_system/jeslogs/${alm.project.name}/${alm.levelRequest.levelName}	Y			*3
ftp.zosconvcmd	quote SITE SBD=(IBM-1047,ISO8859-1)		Y		*4
propsfile.environment	${dir.zosResources}/DEPLOY/environment_deploy.properties	Y			*5
propsfile.objtypes	${dir.zosResources}/globalObjtypes.properties	Y			*6
ftp.active	true;false	Y		Y	*7
undoDeployment	false;true;deletion			Y	

(*1): Phase Models location

(*2: Resource files location

(*3): JES logs location

(*4): FTP subcommand (zOS encoding, transfer encoding) for mainframe

(*5): Environment deploy properties location

(*6): Global Object Types location

(*7): Option to disable the FTP transfer of JCL (useful for troubleshooting)

Build Phases Parameters

In the Build Environment, click on “Edit Phases”, then click on the “Edit Environment Phase Parameter” icon next to the Phases mentioned below.

- **z/OS Copy from Source folder to Target folder**
  Copy components from a Source Environment folder to a Target Environment folder with predefined filters and include/exclude masks using Object Types defined in globalObjtypes property file.

Name	Value	Mandatory	Description
setDeleteComponents	TRUE		*1
setListsComponents	TRUE		*2
setObjectsComponents	TRUE		*3
setPropertyFiles	TRUE		*4
setSourcesComponents	TRUE		*5

(*1): Copy all "sources to be deleted" components to the target.

(*2): Copy Listings object types to the target.

(*3): Copy all Objects object types (i.e. setObjectsZos, setObjectsIdms) to the target.

(*4): Copy Property Files (“*.properties”) to the target.

(*5): Copy all Sources object types (i.e. setSourcesZos, setSourcesIdms) to the target.

- **z/OS Copy Sources to z/OS for compilation (Phase)**
  Select and to copy Copybooks, Includes, Dclgen, Lct, Macro, Map and Program files to their respective z/OS PDS or USS path.

Name	Value	Mandatory	Description
recvend.model	recvend-xmit_jcl.model		*1

(*1): The end JCL model to use after the receive command.

When partitioned data sets are of a different type (PDS vs PDSE) on the build and deploy environments we need to use an intermediate (XMIT) PDS (available in the deploy environment). The Phase will determine the nature (PDS or PDSE) of the partitioned data set it will copy to.

By setting the "xmit Last Level Qualifier" in the "z/OS properties" section for object types with function "LOAD" or "XMIT" in KRC, the Phases are able to identify an intermediate PDS (available in the deploy environment) like the build type. This parameter is not needed if this work-around is not required in your environment.

(note): You can optionally add the ‘save.jeslogs’ parameter with value ‘true’ to a (compilation) Phase, for debugging purposes. The directory for the JES log files is configured as a Build Environment parameter. We advise you to set your Tomcat listings option to true so that the JES logs are visible in your Web browser.

- **z/OS Maps and Programs compilation (Phase)**
  Compiles z/OS Maps in BMS or Sdf2 languages and Programs in mainly Assemble, Cobol, Pli languages with pre-compile of Cics, Databases as Db2, Datacom, IDMS or IMS.

Name	Value	Mandatory	Description
propsfile.languages	Set the location of the “languagesZOS.properties” file*		*1
save.jeslogs	Always save JES logs		

(*1): The languages properties file can be set at the build environment level instead.

Test and Production Phases Parameters
In the Deploy Environment, click on “Edit Phases”, then click on the “Edit Environment Phase Parameter” icon next to the Phases mentioned below.

- **z/OS Promote components and load-modules**

Name	Value	Mandatory	Description
recvend.model	recvend-xmit_jcl.model		*1

(*1): When this is used on the "z/OS Copy Sources to z/OS for compilation" Phase, it will have to be enabled here too.

- **z/OS DB2 Binds transfer and activation**

Name	Value	Mandatory	Description
propsfile.db2Rules	${dir.zosResources}/db2Rules.properties		*1

(*1): To use the DB2 Bind Rules properties.

For a complete list of Phase parameters, visit:
https://docs.kobee.io/integration-zos-phases-catalog/2.1/zOS_Phases_Intro.html

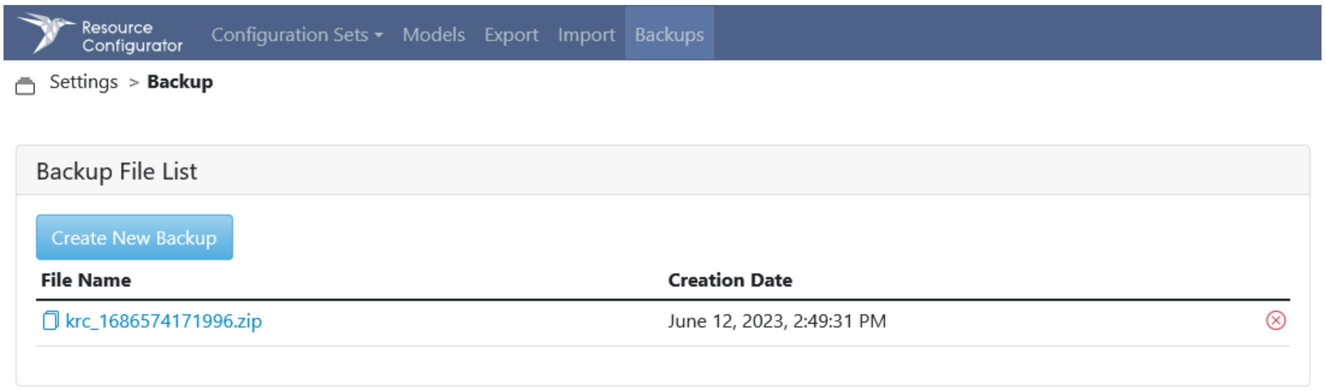
Creating backups and performing imports in KRC

Creating and restoring Configuration Set backups

In the top menu click on “Backups”, next click the “Create New Backup”. A backup contains all the Configuration Sets, but it does not contain any of the actual files you would normally get from an Export action.

To restore the Configuration Sets, on the “Backups” page click the desired backup file and then click the “Restore” button.

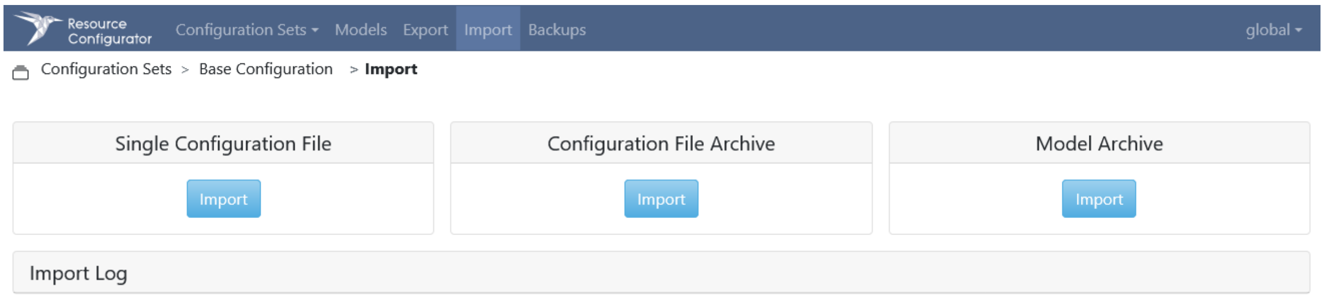
When backups are no longer needed, they can be removed by clicking the red icon at the right side of the file name. KRC will also remove the file from disk.



File Name	Creation Date
krc_1686574171996.zip	June 12, 2023, 2:49:31 PM

Importing files and models

In the top menu click on “Import”, an import action will always target the currently selected Configuration Set, so make sure the correct one is selected by looking at the name in the breadcrumb. There are three ways to import files:



Single Configuration File

Mostly it is faster to import an existing configuration file instead of copying all the original values into a newly created one.

In the popup dialog choose the desired file and make sure to select the correct configuration class as this will determine the import action behavior in KRC. If the file is for a Build or Deploy Environment configuration (the configuration class name will start with “build” or “deploy”) also enter the correct export path (by default this is “BUILD” or “DEPLOY”).

Configuration File Archive

When there are configuration files available, for example from an existing installation, it is possible to import these files as an archive (.zip). Using this type of import will work best in a newly created Configuration Set because there will be no conflicting file names.

It is necessary to add a file list in the form of an XML file to this archive.

In the “dist” folder of KRC there is a “krc_conffiles.zip” archive that is used by KRC for initializing the default Configuration Set, this can also be used as an example on how to write the XML file.

Model Archive

The same information applies as with the “Configuration File Archive” import action.
Again in the “dist” folder there is a “krc_models.zip” archive that can be used as an aid for writing the XML file.

Appendix A

Object types: function types and properties

Function types

Only one function type can be assigned to an Object Type.

Function Type	Description
NONE	No function (technical)
MAP	Source Maps
SRC	Source programs
SRC_LANG	Source Language options
COPY	Copybooks
JCL	JCL's
PROC	JCL Procs
LIST	Listings
BIND	Bind files
LCT	Link control Table
GENERATE	Generated Files
LOAD	Load-modules
DDL	DDL
SQL	SQL
IDMS	IDMS ADS Sources
DATAFILE	DSN or data files
FILE	Other files
XMIT	XMIT files to receive
JOB	Jobs to submit as models

"luw" properties (definitions for Linux, Unix or Windows)

Property in KRC	Exported Property	Description
Luw Acronym	acronym	Acronym (3 to 4 characters)
Path (in VCR)	path	Path for the object (in the VCR)
File Extension	suffix	Associated suffix/extension file (lowercase)
FTP Format	format	ASCII or Binary FTP transfer
FTP Encoding	encoding	FTP data encoding (e.g., ISO8859-1)

“match” properties (Object Types to match for compilation)

Property in KRC	Exported Property	Description
COPY Object Type(s)	copyref	(*1)
LOAD Object Type(s)	loadref	(*2)
LIST Object Type(s)	listref	(*3)
DBRM Object Type(s)	dbrmref	(*4)
GENENERATE Object Type(s)	generef	(*5)
Objects Object Type(s)	objectref	(*6)

(*1): Used for concatenating copybooks and PDSs that contain copybooks. These are used for Object Type(s) with COPY function. (can contain multiple Object Types).

(*2): Used for generated Load-modules. These are used for Object Type(s) with LOAD function (can contain multiple Object Types).

(*3): Used for generated listings. THese are used for Object Type(s) with LIST function (can contain multiple Object Types).

(*4): Used for generated db2 dbrm's. These are used for Object-type with GENERATE function

(*5): Used for other components that are generated. These are used for Object Type(s) with GENERATE function. Except the DBRM Object Type (can contain multiple Object Types).

(*6): Used for managing a list ${objectlibs} of object PDSs in the Bind step. These are used for Object Type(s) with GENERATE function.

Multiple Object Types must be seperated with a comma character.

In case of a double compilation the LOAD and the LIST properties should contain two Object Types each.

“options” properties (to activate actions for the environment Phases)

Property	Value	Description
DB2	YES or NO	
CICS	YES or NO	
IMS	YES or NO	
SAS	YES or NO	
DATACOM	YES or NO	
ORACLE	YES or NO	
JES	YES or NO	
SDF2	COBOL or PLI	
BIND	PLAN or PACKAGE	
NDVR	Type or Language (Free text)	
QMF	FORM, PROC, QUERY	
DEBUG	ABENDAID, IDF, SMARTTEST, XPEDITER	
JMAN	JMODEL (Free text)	
SERVICE	YES or NO	

Property	Value	Description
EXPANDSRC	YES or NO	Expand source
DOUBLE	YES or NO	
CUSTOMER	Free text	Customer CICS value
LANG	XML	Impact Analysis language
LANGMDL	Model to replace the default (in "languagesZOS.properties") model	Extra compile models
LINKMDL	Model to replace the default (in "languagesZOS.properties") model	Extra linkedit models
IDDTYPE	ADSA, DIALOG, IDMSDDDL, MAP, SCHEMA, SUBSCHEMA	IDMS Kobee type
REFTYPE	DDDL, MAPS, SCHEMA, SUBSCH	IDMS reference type
DICTYPE	APPL, DIALOGS, FILE, MAP, MODULE, PROGRAM, PROCESS, RECORD, SCHEMA, SUBSCHEMA, TABLE	IDMS dictionary type

“zos” properties (Definition of DSN properties for the z/OS Object Type)

Property	Description
acronym	Acronym (3 to 4 characters)
dsorg	PO or PS
type	PDSE
lrecl	integer
blksize	integer
recfm	FB,FBA,VB,VBM,U
space	PDS space
unit	Unit location
llq	Last Level Qualifier
xmit	Last Level qualifier for xmit with temporary PDS/PDSE
path	Unix Path in z/OS
suffix	Associated Unix suffix/extension file (lowercase)
getrule	Rule criteria from member name for ftp get selection
getselect	Select criteria from getrule selection
mgetrule	Rule criteria from member name for ftp mget selection
mgetselect	Select criteria from mgetrule selection

“useroptions” (Special options)

Property	Description
loadsuffix	Suffix for map load-module name
suffix	Suffix for service generated source name
buildScript	Build script for starting an external script after compilation
deployScript	Deploy script for starting an external script after JOB execution
debugger	Compile with debugger

Appendix B

Default Project properties

The table below shows if properties are retrieved from Kobee and whether they are changed internally by the Phases (if so, it will be explained in the descriptions). It also shows other global properties that can be set if the default behavior is not adequate. These properties are used everywhere.

Property	Value retrieved from Kobee	Description
${projectAcronym}	${alm.project.name}	*1
${projectRelease}	${alm.project.name}	*2
${objectVersion}		*3
${packageName}	${alm.package.name} (OR ${alm.levelRequest.vcrTag} when 'Release-based')	
${packageNumber}	${alm.package.oid}	*4
${alm.package.extended.oid}	${alm.package.oid}	*5
${project.zos.name}	${alm.project.vcrProjectName}	
${project.zos.release}		*6
${project.zos.package}		*7
${project.zos.packageNumber}		*8
${project.zos.objectVersion}		*9
${buildNumber}	${alm.build.number}	*10
${package.zos.version}		*11
${package.zos.prefix}		*12
${environmentName}	${alm.deploy.environmentName} OR ${alm.build.environmentName}	
${this.requester}	${alm.levelRequest.requester}	*13

(*1): Applies REGEX expression ^(\w{2}).*

(*2): Applies REGEX expresssion .*-(\d*)$ if possible else default (1)

(*3): 1 (is default)

(*4): OR using REGEX expression .*-([0-9]+)$) from ${packageName} if possible else default (1)

(*5): Prepended with 'zero' characters until max length of 7 is reached

(*6): ${projectRelease} formatted using: TEMPLATE (default = R??), prepends with 'R' character followed by zeros

(*7): ${packageNumber} formatted using: TEMPLATE (default = P?????), prepends with 'P' character followed by zeros

(*8): ${packageNumber} formatted using: TEMPLATE (default = ??????), prepends with zeros

(*9): ${objectVersion} formatted using: TEMPLATE (default = 1.00${projectRelease}.???), prepends with '1.00${projectRelease}.' and zeros

(*10): Prepends ${alm.build.number} with 'B' character followed by zeros until a max length of 8 is reached

(*11): ${project.zos.name}_${project.zos.release}_${project.zos.package}

(*12): ${project.zos.name}_${project.zos.release}_${project.zos.package}

(*13): Uppercase format

Changing templates and patterns

The table below shows the syntax to add a custom parameter in the Default Project properties so that the templates and patterns can be customized.

Property template or pattern	Default value
project.zos.release.template	R??
project.zos.package.template	P?????
project.zos.packageNumber.template	??????
project.zos.objectVersion.template	1.00${projectRelease}.???
buildNumber.template	B???????
projectAcronym.pattern	^(\w{2}).*
projectRelease.pattern	.*-(\d*)$
packageNumber.pattern	.*-([0-9]+)$

Special case properties

The Job Name property: ${jobname} is generated from the Userid z/OS (zos.userid) property defined in the Build and Deploy Environment Configuration.

(Note: It is initiated by the Username (userid) parmsFTPZOS properties file and completed with a phase suffix depending on the Phase in course. The zosCompilation Phase increments this phase-suffix for managing several JCLs on JES together.)

Appendix C

Practical example on how properties are used in the models

1. Compile model sample

```
// SET PARMCOB='${compilation.lang.parms}'
// SET PARMCOB0='${pgm.compile.parms.batch}'
//*******************************************************************
//**    COMPILE THE ELEMENT ${ref.acronym}.${member}
//*******************************************************************
//${pgm.language}  EXEC PGM=${compilation.lang.program},COND=(${compilation.lang.rcmax},LT),
//   PARM=(&PARMCOB,
//   &PARMCOB0,NOMDECK),MAXRC=${compilation.lang.rcmax},WARNING=${compilation.lang.warning}
//STEPLIB  DD  DISP=SHR,
//             DSN=${compilation.lang.prefix}.${compilation.lang.lib}
//SYSIN    DD  DISP=(OLD,PASS),DSN=&&&SRCOMPIL
//SYSLIN   DD  DISP=(,PASS),DSN=&&OBJECT,
//             UNIT=SYSDA,SPACE=(CYL,(2,2)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT8   DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT9   DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT10  DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT11  DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT12  DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT13  DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT14  DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSUT15  DD  UNIT=SYSDA,SPACE=(CYL,(5,3))
//SYSPRINT DD  DISP=(MOD,PASS),DSN=&&COMPLIST,
//            UNIT=SYSDA,SPACE=(TRK,(10,90),RLSE)
//*            DCB=(RECFM=FBA,LRECL=133,BLKSIZE=0)
//*
//SYSMDECK DD DISP=(MOD,PASS),DSN=&&SYSMDECK,
//            UNIT=SYSDA,SPACE=(TRK,(10,90),RLSE)
//*            DCB=(RECFM=FBA,LRECL=133,BLKSIZE=0)
//SYSLIB   DD DISP=(,PASS),DSN=&&SYSLIBCP,
//            UNIT=SYSALLDA,SPACE=(TRK,(1,1,1),RLSE),
//            DCB=(DSORG=PO,RECFM=FB,LRECL=80,BLKSIZE=27920)
${copylibs}
${extendedCopylibs}
```

Generated properties

The generated “${ref.acronym}” property corresponds with the MAP/SRC object type “zos=acronym:xxx”.

The generated “${member}” property contains the member name (in uppercase with a max. length of 8 characters) coming from the source name.

For a compilation process, models may have the generated values of properties “${copylibs}” or “${incllibs}”.

The PDS names for the SYSLIB for all “copyref” object-types are concatenated. The SYSLIB DDN and DSN first card must be defined in the model(s).

The concatenation is defined in the **environment_build_zos** property file with “env.lib.level.compile.n” and/or “env.path.level.compile.n“ properties.

```
env.lib.level.compile.0=${env.lib}
env.lib.level.compile.0.suffix=
env.lib.level.compile.1=${env.prefix1}.${env.qualifA}
env.lib.level.compile.1.suffix=
env.lib.level.compile.2=${env.prefix2}.${env.qualifB}
env.lib.level.compile.2.suffix=
env.lib.level.compile.3=${env.prefix3}.${env.qualifC}
env.lib.level.compile.3.suffix=
env.lib.level.compile.4=${env.zos.prefix}.${env.qualifD}
env.lib.level.compile.4.suffix=
```

“${extendedCopylibs}” uses the “extendedCopylibs_jcl.model” with the substituted properties:

```
//${ignore.assemble}        DD  DISP=SHR,DSN=${compilation.lang.prefix}.${compilation.lang.copylib}
//${ignore.cics}          DD DISP=SHR,DSN=${cics.prefix}.${cics.lang.copylib}
//*${ignore.db2}          DD DISP=SHR,DSN=${db2.linklib.prefix}.${db2.lang.copylib}
//*${ignore.assemble}         DD DISP=SHR,DSN=${env.prefix}.PROD.${compilation.lang.copylib}
//*${ignore.cobol}          DD DISP=SHR,DSN=${env.prefix}.PROD.${compilation.lang.copylib}
//*${ignore.pli}          DD DISP=SHR,DSN=${env.prefix}.PROD.${compilation.lang.copylib}
```

2. Link-edit model sample

```
//*******************************************************************
//**   LINKEDIT PROGRAM                                            **
//*******************************************************************
//      SET PARMLNK='${linkedit.lang.parms}'
//      SET LINKOPT='${pgm.link.parms.batch}'
//LKEDIT   EXEC PGM=${linkedit.lang.program},COND=(${linkedit.lang.rcmax},LT),
//    PARM='&PARMLNK',MAXRC=${linkedit.lang.rcmax}
//*    PARM=(&PARMLNK,&LINKOPT),MAXRC=${linkedit.lang.rcmax}
//SYSLMOD   DD DISP=SHR,
//             DSN=${linklib1}(${pgm.loadname})
//SYSDEFSD DD DUMMY
//SYSPRINT DD  DISP=(MOD,PASS),DSN=&&LINKLIST,
//             UNIT=VIO,SPACE=(TRK,(10,10)),
//             DCB=(RECFM=FBA,LRECL=122,BLKSIZE=0)
//SYSLIB   DD  DISP=(NEW,DELETE,DELETE),DSN=&&SYSLIBLK,
//              UNIT=${env.zos.unit},SPACE=(TRK,(1,1,1)),
//              DCB=(DSORG=PO,RECFM=FB,LRECL=80,BLKSIZE=0)
${linklibs}
${extendedLinklibs}
```

Generated properties

A link-edit and other models may contain following generated values of properties:

- "${lctlibs}" contains generated PDS libs in concatenation used in "copyLct_jcl.model", if LCT function is activated.
- "${linklib}" contains generated PDS of first loadref lib.
- "${linklib[i]}" contains generated PDS of each loadref object-type in desired order.
- "${linklibs}" contains generated PDS of SYSLIB for all loadref object-types in concatenation.

The concatenation is managed in the **environment_build_zos.properties** file (or similar) by the "env.lib.level.link.n" properties.

- "${objectlibs}" contains generated PDS of SYSLIB objectref object-types in concatenation
- "${extendedLinklibs}" uses the extendedLinklibs_jcl model with the substituted properties
- "${listlib}" contains generated PDS of first listref object-type
- "${listlib[i]}" contains generated PDS of each listref object-type in indicated order (i).
- "${genlibs}" contains generated PDS of SYSLIB generef object-types in indicated order (i).
- "${[objtype].ignorelib}" and "${[objtype].ignorepath}" provide a star "*"
  if the z/OS file is available on a PDS or on a Unix folder, except for LIST function object-types.
- "${listing.ignorelib}" and "${listing.ignorepath}" provide a star "*"
  if the z/OS file is available on a PDS or on a Unix folder. That is depending on the object-type "zos" line where the "path" is defined or not.

LOAD function object-types:

- “${[objtype].acronym}” contains “zos=acronym:xxx” of LOAD object-type
- “${acronym[i]}” contains “zos=acronym:xxx” of object-type [i]
- “${loadlib[i]}” contains PDS (using llq) or full Unix path of object-type [i]
- “${xmitfile[i]}” contains PDS (using xmit) of object-type [i]
- “${pgm.loadname[i]}” contains source load-module names [i]
- "${pgm.prodname[i]}" contains source load-module names[i] for Production deployment.

For LIST and GENERATE and DBRM function object-types:

- “${[objtype].acronym}” contains “zos=acronym:xxx” of object-type
- “${[objtype].zos.llq}” or “${[objtype].zos.path}” contains the llq or path of object-type
- “${genlib_[objtype]}” contains PDS or full Unix path of object-type
- For DBRM function object-type with DB2:
- “${objtype.dbrm}” contains the dbrm object-type
- “${dbrmlib}” contains PDS of dbrm object-type

All “${pgm…}“ properties as “${pgms…}” properties in defaultPgms property file.

Other default generated properties are defined in the Default Project properties fle as mentioned Appendix B.

IKAN Development
Motstraat 30
2800 Mechelen, Belgium
Tel. +32 15 238427

info@kobee.io
www.kobee.io