

Deployit Packaging Manual

Version 3.7.4

Table of Contents

Table of Contents	2
Preface	3
Introduction	3
Packages	3
DAR Format	3
Manifest Format	3
Specifying the Application	3
Specifying Artifacts	4
Specifying Resource Specifications	4
Setting Complex Properties	4
Properties referring to other CIs	4
Set of strings properties	5
List of strings properties	5
Map of string to string properties	5
Boolean and enumeration properties	5
Using Placeholders in CI properties	5
Scanning for placeholders in Artifacts	5
Making a deployment package	6
Making a Deployment Package by Hand	6
Using the Deployit Maven Plugin	7
Using the Maven jar Plugin	7
Using the Ant jar Task	8
Sample Packages	8

Preface

This manual describes how to package applications for use in Deployit.

See the **Deployit Reference Manual** for background information on Deployit and deployment concepts.

Introduction

The Deployit deployment automation tool is designed to help you deploy application packages to target middleware. To make it possible to deploy your applications, they must be packaged in a format that Deployit understands. This manual describes Deployit's standard package format, the *Deployment ARchive* (DAR) format and various other topics related to packaging applications.

Deployit also offers the plug points to implement a custom importer that recognizes a different format.

Packages

Deployit uses the *Unified Deployment Model (UDM)* to structure its deployments (see the **Deployit Reference Manual** for more information). In this model, deployment packages are containers for complete application distribution, that include both the application artifacts (EAR files, static content) as well as the resource specifications (datasources, topics, queues, etc.) that the application needs to run.

Packages should be independent of the target environment and contain customization points (for instance placeholders in configuration files) that supply environment-specific values to the deployed application. This enables a single artifact to make the entire journey from development to production.

DAR Format

Out of the box, Deployit supports its own DAR format for packages. A valid DAR package has the following characteristics:

1. It's a ZIP archive.
2. It has a manifest file in `META-INF/MANIFEST.MF` containing a description of the contents of the package.

Valid DAR archives can be produced using standard command line tools, such as `zip`, the Java `jar` utility, the Maven `jar` plugin or the Ant `jar` task. There is also a Deployit maven plugin to facilitate packaging. See the section **Using the Maven plugin** below.

In addition to packages in a compressed archive format, Deployit can also import *exploded* DARs or archives that have been extracted.

Manifest Format

The manifest file included in a DAR describes the contents of the archive for Deployit. When importing a package, the manifest is used to construct CIs in Deployit's repository based on the contents of the imported package. For background information on the JAR format and manifest, see the [JAR file specification](#).

A valid Deployit manifest starts with the following preamble:

```
Manifest-Version: 1.0
Deployit-Package-Format-Version: 1.3
```

This identifies the Java manifest version and the Deployit package format version.

Specifying the Application

A deployment package contains a specific version of an application. These entries tell Deployit that the package contains the *AnimalZoo* application version *4.0*:

```
CI-Application: AnimalZoo-ear
CI-Version: 4.0
```

These entries are part of the manifest preamble.

Specifying Artifacts

Artifacts are represented by files or folders in the deployment package. To import an artifact as a CI in Deployit, use:

```
Name: AnimalZooBE-1.0.ear
CI-Type: jee.Ear
CI-Name: AnimalZooBE
```

The standard `Name` entry must refer to an actual file included in the DAR. The `CI-Type` specifies a CI type name that is available in the system. The required `CI-Name` property indicates the name of the CI to be created.

Similarly, this construct can be used to create a folder CI:

```
Name: conf
CI-Type: file.Folder
CI-Name: configuration-files
```

Specifying Resource Specifications

Resource specifications are not represented by files in the deployment package. They represent resources that must be created on the target middleware when the application is deployed. Resource specifications are constructed completely based on the information in the manifest. The manifest also specifies values for properties of the CI.

For example, this manifest snippet constructs a *was.OracleDataSourceSpec* CI:

```
Name: petclinicDS
CI-Type: was.OracleDataSourceSpec
CI-driver: com.mysql.jdbc.Driver
CI-url: jdbc:mysql://localhost/petclinic
CI-username: petclinic
CI-password: my$ecret
```

The `Name` entry specifies the name of the CI to be created. In contrast to the manifest specification, the `Name` property in a Deployit manifest does not refer to a physical file present in the DAR in the case of a resource specification. The `CI-Type` specifies a CI type that is available in the system.

Note: the names of artifacts in your package must conform to platform requirements. For instance, a *file.Folder* CI with name "q2>2" cannot be deployed to a Windows host, because ">" may not be part of a file or directory name in Windows.

The other entries, `CI-url`, `CI-username` and `CI-password` refer to properties on the datasource CI. These properties will be set to the values specified. In general, any property on a CI can be set using the `CI-<propertyname>` notation. See the **Command Line Interface (CLI) Manual** for information or how to obtain the list of properties that a particular CI type supports, or consult the relevant CI reference documentation.

Note that it is also possible to add resource specifications to a package that is already imported in Deployit. See the **Command Line Interface (CLI) Manual** for more information.

Setting Complex Properties

The above example showed how to set string properties to a certain value. In addition to strings, Deployit also supports references to other CIs, sets of strings, maps of string to string, booleans and enumerations. Here are some examples.

Properties referring to other CIs

```
Name: myResourceSpecName
...

Name: my-artifact-file-name.ext
CI-Name: myArtifactCiName
...

...
CI-artifactRefProperty: myArtifactCiName
CI-resourceSpecRefProperty: myResourceSpecName
```

```
...
CI-setOfCIProperty-EntryValue-1: myResourceSpecName
CI-setOfCIProperty-EntryValue-2: myArtifactCiName
...
CI-listOfCIProperty-EntryValue-1: myOtherResourceSpecName
CI-listOfCIProperty-EntryValue-2: myOtherArtifactCiName
```

Note that references to artifacts use the referent's **CI-Name** property and references to resource specifications use the referent's **Name** property.

Set of strings properties

```
CI-setOfStringProperty-EntryValue-1: a
CI-setOfStringProperty-EntryValue-2: b
```

List of strings properties

```
CI-listOfStringProperty-EntryValue-1: a
CI-listOfStringProperty-EntryValue-2: b
```

Map of string to string properties

```
CI-mapOfStringToStringProperty-key1: value1
CI-mapOfStringToStringProperty-key2: value2
```

Boolean and enumeration properties

```
CI-boolProperty: true
CI-boolProperty: false
CI-enumProperty: ENUMVALUE
```

Using Placeholders in CI properties

Deployit supports the use of *placeholders* to customize a package for deployment to a specific environment. CI properties specified in a manifest file can also contain placeholders. These placeholders are resolved from *dictionary* CIs during a deployment (see the **Deployit Reference Manual** for an explanation of placeholders and dictionaries). This is an example of using placeholders in CI properties in a *was.OracleDatasourceSpec* CI:

```
Name: petclinicDS
CI-Type: was.OracleDatasourceSpec
CI-driver: com.mysql.jdbc.Driver
CI-url: jdbc:mysql://localhost/petclinic
CI-username: {{DB_USERNAME}}
CI-password: {{DB_PASSWORD}}
```

Deployit also supports an alternative way of using dictionary values for CI properties. If the dictionary contains keys of the form *deployedtype.property*, these properties are automatically filled with values from the dictionary (provided they are not specified in the deployable). This makes it possible to use dictionaries without specifying placeholders. For example, the above could also have been achieved by specifying the following keys in the dictionary:

```
was.OracleDatasource.username
was.OracleDatasource.password
```

Scanning for placeholders in Artifacts

By default Deployit will try to scan files inside packages for the presence of placeholders. These will be added to the *placeholders* field in the artifact, so that they can be replaced upon deployment of said package. However there are scenarios when you want to disable this behavior. This is triggered by setting the *scanPlaceholders* flag on an artifact.

```
Name: sample.txt
CI-Type: file.File
CI-Name: sample
CI-scanPlaceholders: false
```

To avoid scanning of binary files, only files with the following extensions are scanned:

```
cfg, conf, config, ini, properties, props, txt, asp, aspx,
htm, html, jsf, jsp, xht, xhtml, sql, xml, xsd, xsl, xslt
```

You can change this list by setting the `textFileNamesRegex` property on the artifact. Not that it takes a regular expression.

If you want to enable placeholder scanning, but the package contains several files that should **not** be scanned, use the `excludeFileNamesRegex` property on the artifact:

```
Name: petclinic-1.0.ear
CI-Type: jee.War
CI-Name: petclinic
CI-excludeFileNamesRegex:.*\.properties
```

Making a deployment package

To illustrate the way a deployment package can be created, let's describe a complete example of how to do this by hand. Subsequent sections will describe how to automate this by using the Deployit maven plugin, the maven `jar` plugin or the ant `jar` task.

Making a Deployment Package by Hand

Let's say we want to create a package for version 1.0 of our brand new PetClinic application. The application contains an EAR file with the application code, a configuration folder containing configuration files and a datasource. Start by creating a directory `petclinic-package` to hold the package content:

```
mkdir petclinic-package
cd petclinic-package
```

Now, collect the EAR file and configuration directory and store them in the newly created directory:

```
cp /some/path/petclinic-1.0.ear .
cp -r /some/path/conf .
```

The datasource is a resource specification, not an artifact, so there is no file to include.

Now, let's create the DAR manifest for these entries. After the required preamble, add the application and version

```
CI-Application: PetClinic
CI-Version: 1.0
```

Add the EAR and configuration folder:

```
Name: petclinic-1.0.ear
CI-Type: jee.Ear
CI-Name: PetClinic-Ear

Name: conf
CI-Type: file.Folder
CI-Name: PetClinic-Config
```

Add the datasource to the manifest as follows:

```
Name: PetClinic-ds
CI-Type: was.OracleDataSourceSpec
CI-driver: com.mysql.jdbc.Driver
CI-url: jdbc:mysql://localhost/petclinic
CI-username: {{DB_USERNAME}}
CI-password: {{DB_PASSWORD}}
```

Note how the datasource uses placeholders for username and password.

The complete manifest looks like this:

```

Manifest-Version: 1.0
Deployit-Package-Format-Version: 1.3
CI-Application: PetClinic
CI-Version: 1.0

Name: petclinic-1.0.ear
CI-Type: jee.Ear
CI-Name: PetClinic

Name: conf
CI-Type: file.Folder
CI-Name: PetClinic-Config

Name: PetClinic-ds
CI-Type: was.OracleDataSourceSpec
CI-driver: com.mysql.jdbc.Driver
CI-url: jdbc:mysql://localhost/petclinic
CI-username: {{DB_USERNAME}}
CI-password: {{DB_PASSWORD}}

```

Save the manifest outside of the package directory, so for instance in the `/tmp` directory. Finally, create the DAR archive with the command:

```
jar cmf ../petclinic-1.0.dar /tmp/MANIFEST.MF *
```

The resulting archive can be imported into Deployit.

Using the Deployit Maven Plugin

To enable continuous deployment, Deployit can be integrated with the Maven build system. The Deployit Maven plugin enables you to integrate Maven and Deployit. Specifically, the plugin supports:

- Creating a deployment package containing artifacts from the build
- Performing a deployment to a target environment
- Undeploying a previously deployed application

For more information, see [the Deployit maven plugin documentation](#).

Using the Maven jar Plugin

The standard maven `jar` plugin can also be used to create a Deployit package.

- Create a manifest file conforming to the Deployit manifest standard (see section **Manifest Format** above).
- Create a directory structure containing the files as they should appear in the package.

In the maven POM, configure the jar plugin as follows (the manifest file is assumed to be in the project's `src/main/resources/META-INF` directory): In the maven POM, configure the jar plugin as follows (the manifest file is assumed to be in the project's `src/main/resources/META-INF` directory):

```

<project>
...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      ...
      <configuration>
        <includes>
          <include>**/*</include>
        </includes>
        <archive>
          <manifestFile>src/main/resources/META-
INF/MANIFEST.MF</manifestFile>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
...
</project>

```

The Deployit package can be generated by invoking the command:

```
mvn package
```

Using the Ant jar Task

Creating a Deployit package via Ant is possible using the `jar` task.

- Create a manifest file conforming to the Deployit manifest standard (see section **Manifest Format** above).
- Create a directory structure containing the files as they should appear in the package.

In the Ant build file, include a `jar` task invocation as follows (the manifest file is assumed to be called `MANIFEST.MF`):

```
<jar destfile="package.jar"
    basedir="."
    includes="**/*"
    manifest="MANIFEST.MF"/>
```

Sample Packages

Deployit ships with several sample packages. These examples are stored in the Deployit server's `importablePackages` directory.