Liferay Cloning Tool

Purpose

Liferay Cloning Tool is an OSGi based application that offers administrators an easy way to prepare a Liferay environment for development or test purposes after duplicating it from a production environment. The goal is to avoid any unintentional effects on production systems caused by a development/test environment, to hide potentially sensitive data from developers and also to make the system easier to use for them.

By default it has 5 built-in steps:

- Updating user passwords to let developers use specific accounts (especially Administrator)
- Deleting password policies so e.g. they won't expire
- Updating remote staging data to avoid publishing to an actual remote live server from a development environment
- Updating user data to prevent developers from seeing e.g. the Skype ID or the address of the users of the production portal
- Updating virtual hosts to avoid collisions with the actual production system's virtual hosts

Being a modular OSGi application, the Cloning Tool can be easily customized or extended by custom processes besides the built-in ones.

It can be configured and executed in 2 ways, either manually or by using the UI of LiferayUp. These will be detailed in the following chapters.

How to manually configure, deploy and run?

The first way to run the tool is the manual one. When using this mode, you have to ensure that the duplication from the production environment was done previously. However, you should NOT start up the duplicated bundle before configuring and deploying the Cloning Tool. Here's how you can do it:

1. You should first set up a configuration file for the tool. The name of the file should look like *<full_path_of_configuration_class>-<companyId_or_default>.cfg* This strict format is required by the configuration framework that we are using for OSGi modules. In the case of the Cloning Tool this must be `com.liferay.cloning.configuration.CloningConfiguration-default.cfg`

Within this file you should manually define the configuration options for the tool using a *key=value* format. The available configuration options are listed in detail in the next section of this document. An example for the content of the configuration file:

```
passwordCloningUpdaterUpdatePasswords=true
passwordCloningUpdaterUserIds=
passwordCloningUpdaterNewPassword=welcome

passwordPolicyCloningUpdaterDeletePasswordPolicies=true

stagingDataCloningUpdaterUpdateStagingData=true
stagingDataCloningUpdaterOldRemoteHosts=productionhost1.com,p
roductionhost2.com
stagingDataCloningUpdaterOldRemotePorts=8080,8080
stagingDataCloningUpdaterOldRemoteGroupIds=12345,67890
stagingDataCloningUpdaterNewRemoteHosts=devhost1.com,devhost2
.com
stagingDataCloningUpdaterNewRemotePorts=8080,8080
stagingDataCloningUpdaterNewRemoteGroupIds=12345,67890

userDataCloningUpdaterUpdateUserData=false

virtualHostCloningUpdaterUpdateVirtualHosts=true
virtualHostCloningUpdaterOldVirtualHosts=productionvirtualhos
t.com
virtualHostCloningUpdaterNewVirtualHosts=devvirtualhost.com
```

2. When the configuration file is ready, you should place it in the `osgi/configs` directory of your bundle's home.

3. The Cloning Tool by default consists of 2 OSGi modules that reside in 2 .jar files: `com.liferay.cloning.api.jar` and `com.liferay.cloning.service.jar` . You should place these files in the deploy folder of your bundle's home.

4. Now you can run the tool by simply starting up the portal, and if the Cloning Tool either hasn't been executed on this bundle, or its last execution did not complete successfully, it will be automatically executed during startup.

That's it. If all the steps of the tool complete successfully, then your development environment should be ready to use.

As mentioned above, if the execution was successful the Cloning Tool is not going to run during any consecutive startups. There can be cases though, when you would want the tool to run again even if it completed successfully on the current environment before. In order to

achieve this, you should set forcedExecution=true in the configuration file described in step 1. above.

You should keep in mind in this case, that a forced execution will run the entire tool again, so if you would like to rerun specific cloning steps only, then you should also set the corresponding option to false for the rest of them (e.g. passwordCloningUpdaterUpdatePasswords=false).

Configuration options

See the available built-in configuration options described below. These can be used when creating the configuration file in step 2. of the previous section.

forcedExecution
Defines whether the cloning steps should be executed regardless of any previous executions on the same environment.
Default value: false

cloningStepsMaxNumber
Defines the maximum number of steps that can be defined for the Cloning Tool. It's needed in order to be able to prioritize the steps but also avoid an infinite loop when running them.
Default value: 100

baseCloningUpdaterBatchSize
Defines the size of a batch of objects fetched at once when running cloning steps that operate on a large number of entities using ActionableDynamicQueries. It can be adjusted based on the system resources of the hosting environment.
Default value: 500

passwordCloningUpdaterUpdatePasswords
Defines if the built-in cloning step for updating user passwords should be executed.
Default value: true

passwordCloningUpdaterUserIds
A comma separated list of userIds for which the passwords should be updated. If passwordCloningUpdaterUpdatePasswords=true but there are no userIds listed for this option, then all user's passwords will be updated.

passwordCloningUpdaterNewPassword
The new password that should be set when executing the built-in cloning step for updating user passwords.

passwordPolicyCloningUpdaterDeletePasswordPolicies

Defines if the built-in cloning step for deleting password policies should be executed.
Default value: true

stagingDataCloningUpdaterUpdateStagingData
Defines if the built-in cloning step for updating staging data should be executed.
Default value: true

stagingDataCloningUpdaterOldRemoteHosts
A comma separated list of the remote staging live hostnames that should be updated.

stagingDataCloningUpdaterOldRemotePorts
A comma separated list of the remote staging live port numbers that should be updated.
WARNING: The number of port numbers in this list must match the number of hostnames set for stagingDataCloningUpdaterOldRemoteHosts and if you are configuring manually, you should take care of the right order!

stagingDataCloningUpdaterOldRemoteGroupIds
A comma separated list of the remote staging live groupIds that should be updated.
WARNING: The number of grouIds in this list must match the number of hostnames set for stagingDataCloningUpdaterOldRemoteHosts and if you are configuring manually, you should take care of the right order!

stagingDataCloningUpdaterNewRemoteHosts
A comma separated list of the new remote staging live hostnames that should be set.
WARNING: The number of hostnames in this list must match the number of hostnames set for stagingDataCloningUpdaterOldRemoteHosts and if you are configuring manually, you should take care of the right order!

stagingDataCloningUpdaterNewRemotePorts
A comma separated list of the new remote staging live port numbers that should be set.
WARNING: The number of port numbers in this list must match the number of hostnames set for stagingDataCloningUpdaterOldRemoteHosts and if you are configuring manually, you should take care of the right order!

stagingDataCloningUpdaterNewRemoteGroupIds
A comma separated list of the new remote staging live groupIds that should be set.
WARNING: The number of groupIds in this list must match the number of hostnames set for stagingDataCloningUpdaterOldRemoteHosts and if you are configuring manually, you should take care of the right order!

userDataCloningUpdaterUpdateUserData
Defines if the built-in cloning step for updating user data should be executed.
Default value: true

virtualHostCloningUpdaterUpdateVirtualHosts
Defines if the built-in cloning step for updating virtual hosts should be executed.

Default value: true

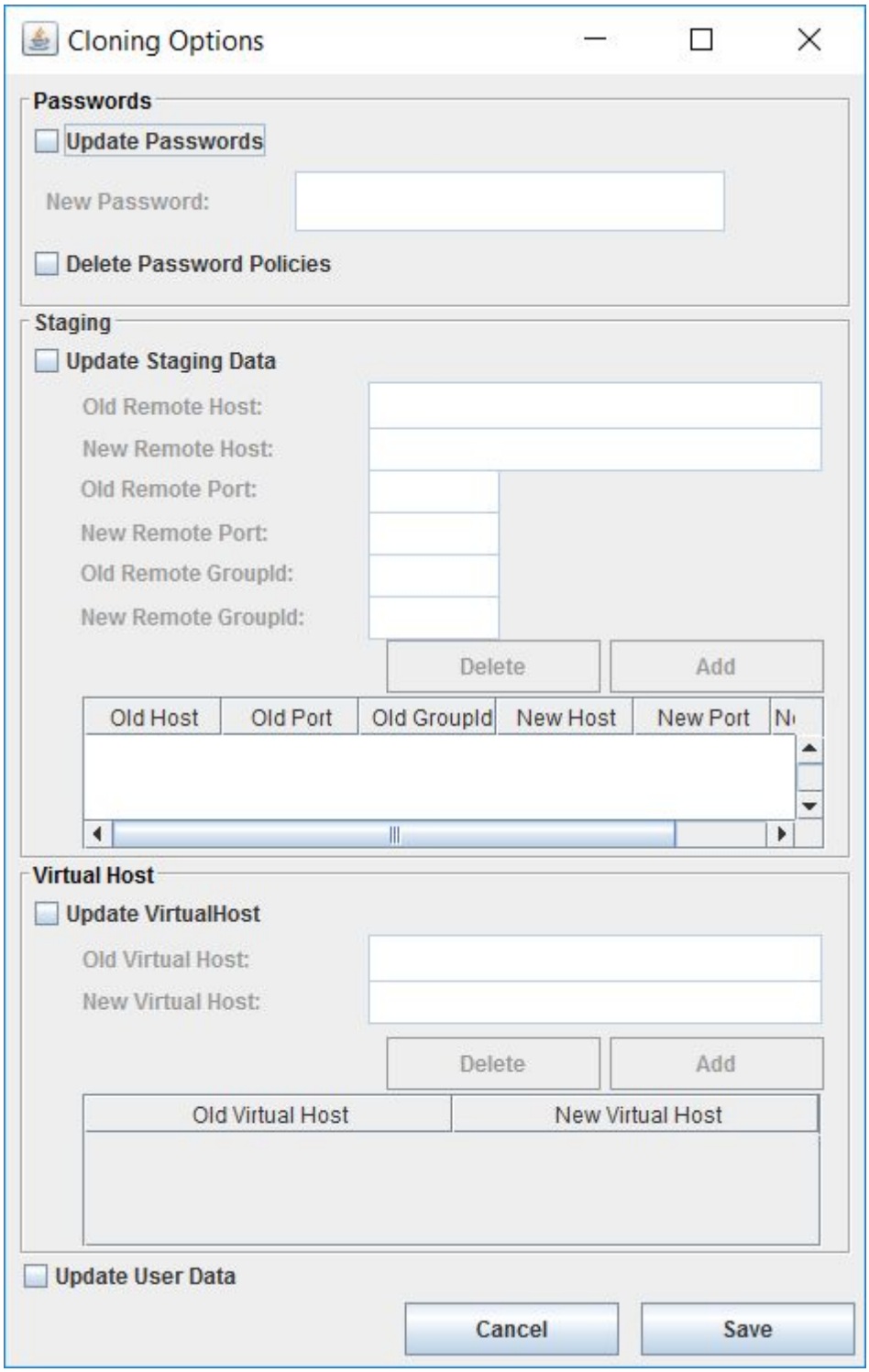
virtualHostCloningUpdaterOldVirtualHosts
A comma separated list of the virtual hosts that should be updated.

virtualHostCloningUpdaterNewVirtualHosts
A comma separated list of the new virtual hosts that should be set.
WARNING: The number of virtual hosts in this list must match the number of virtual hosts set for virtualHostCloningUpdaterOldVirtualHosts and if you are configuring manually, you should take care of the right order!

How to run with LiferayUp?

You have a much easier job if you use LiferayUp to set up the development environment. The next version of LiferayUp will be able to not only put together the bundle, deploy the plugins and install the patches needed, but also to import a dump into the database. This means you will have the duplicated production system ready.

In order to make Cloning Tool easier to use, it will be able to configure and deploy it from within the next version of LiferayUp as well. The configurations will be defined on the UI and LiferayUp will generate the config file based on your input. See a screenshot of a prototype of the planned UI:



When you execute LiferayUp and it finishes building your environment, it will also copy the Cloning Tool's OSGi modules and configuration file into the bundle's appropriate directories. So you literally only need to start the portal for the first time, and the Cloning Tool is going to run upon startup with your defined configurations.

Modules and classes

By default the Cloning Tool consists of 2 OSGi modules that have the following packages and classes/interfaces:

liferay-cloning-api

```
com.liferay.cloning.api
      CloningConstants
      CloningException
      CloningStep
```

liferay-cloning-service

```
com.liferay.cloning.configuration
      CloningConfiguration
com.liferay.cloning.executor
      CloningExecutor
com.liferay.cloning.updater
      BaseCloningUpdater
      PasswordCloningUpdater
      PasswordPolicyCloningUpdater
      StagingDataCloningUpdater
      UserDataCloningUpdater
      VirtualHostCloningUpdater
```

`CloningStep` is the base interface for cloning steps, and all steps has to implement it in order to be able to run. `BaseCloningUpdater` implements it, but other than that it's basically empty, it just offers a place to add anything that we want to inherit to it's extending classes. The rest of the updaters extend `BaseCloningUpdater`.

`CloningConfiguration` is the interface where the methods representing the configuration options are declared, and `CloningExecutor` is the controller class that invokes the cloning steps at portal startup.

You can check the source code at https://github.com/giros/liferay-cloning

How to extend?

The most interesting part is how can you add custom cloning steps to be executed by the tool?

First of all you will need to create a new module. This can be done either with Liferay IDE or the Blade command line tool. Detailed descriptions about how to create modules with these can be found on the developer network:
https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/liferay-ide
https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/blade-cli
I recommend using the IDE as it has some features that I'm still not sure how to invoke from Blade.

When your module has been created, it's important to import the built-in packages by adding the following in your module's *bnd.bnd* file:

```
Export-Package:\
      com.liferay.cloning.api,\
      com.liferay.cloning.configuration,\
      com.liferay.cloning.executor,\
      com.liferay.cloning.updater
```

This ensures that you can use the built-in classes in your custom code. Next you need to add the dependency on the built-in modules to your module's *build.gradle* file:

```
compile project(":modules:liferay-cloning-api")
compile project(":modules:liferay-cloning-service")
```

Now you're ready to create the class of your new cloning step. There are a couple of things that you should ensure to make it work properly:

- It has to implement the `CloningStep` interface
- It has to be an OSGi service component of the `CloningStep` service. This means you have to define `service = {CloningStep.class}` in the `@Component` annotation of your class.
- You have to set a priority for your cloning step that defines when it is executed in between the rest of the steps. The priority is an integer and the built-in steps by default has priorities as 10, 20, 30, 40 and 50 (see their source code). You can set the priority of your new cloning step as a property of the `@Component` annotation, e.g.: `property = {"cloning.step.priority=60" }`
- Now you can develop your new cloning step by implementing its `execute()` method

See an example of the new cloning step class below:

```
package com.liferay.cloning.example;

import com.liferay.cloning.api.CloningStep;

import java.util.Map;
```

```
import org.osgi.service.component.annotations.Component;

@Component(
	property = {"cloning.step.priority=60"},
	service = {CloningStep.class, ExampleCloningStep.class},
)
public class ExampleCloningStep implements CloningStep {

	@Override
	protected void execute() throws Exception {

		// Actual code of the cloning step

	}

}
```

Usually you will want to be able to configure your new cloning step by defining it's own configuration options. In order for this you need to do the following:

- Create a new configuration interface
- Annotate the interface definition with `@Meta.OCD`, which has to have an id parameter with a value that is commonly the fully qualified name of the interface. E.g.: `id = "com.liferay.cloning.example.configuration.ExampleConfiguration"`
- Add methods to this interface, where the name of the method will be the actual configuration option key, and its return type will be the type of the value that can be assigned to the option. E.g: `public boolean exampleConfigurationOption();`
- These methods should be annotated with `@Meta.AD`. This annotation can have parameters also, the most common is probably the default value that is used if we don't specify anything in the .cfg file. E.g: `@Meta.AD(deflt = "true", required = false)`

Here is an example configuration interface:

```
package com.liferay.cloning.example.configuration;

import aQute.bnd.annotation.metatype.Meta;

@Meta.OCD(
	id =
"com.liferay.cloning.example.configuration.ExampleConfiguration",
	name = "example.configuration.name"
)
```

```
public interface CloningConfiguration {

	@Meta.AD(deflt = "true", required = false)
	public boolean exampleConfigurationOption();

}
```

- Once your configuration interface has been created, you need to modify the class of your cloning step to use it. First of all you have to extend the `@Component` annotation again to set the id of your configuration that you defined at the first step above. E.g.: `configurationPid = "com.liferay.cloning.example.configuration.ExampleConfiguration"`
- Then you have to add a private field to your class with the type of your configuration interface e.g: `private ExampleConfiguration _exampleConfiguration;`
- And you need a method that reads the configuration into this field:

```
@Activate
@Modified
protected void readConfiguration(Map<String, Object>
properties) {
	_exampleConfiguration= Configurable.createConfigurable(
		ExampleConfiguration .class, properties);
}
```

  It's important that it should have the @Activate and @Modified annotations.
- After this, you can use the configuration option values by simply invoking the corresponding method e.g.: `_exampleConfiguration.exampleConfigurationOption();`

And the example cloning step class extended:

```
package com.liferay.cloning.example;

import com.liferay.cloning.api.CloningStep;
import
com.liferay.cloning.example.configuration.ExampleConfiguration;

import aQute.bnd.annotation.metatype.Configurable;

import java.util.Map;

import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Modified;

@Component(
```

```
	property = {"cloning.step.priority=60"},
	service = {CloningStep.class, ExampleCloningStep.class},
	configurationPid =

"com.liferay.cloning.example.configuration.ExampleConfiguration"
)
public class ExampleCloningStep implements CloningStep {

	@Override
	protected void execute() throws Exception {

		// Actual code of the cloning step

	}

	@Activate
	@Modified
	protected void readConfiguration(Map<String, Object>
properties) {
		_exampleConfiguration = Configurable.createConfigurable(
			ExampleConfiguration.class, properties);
	}

	private ExampleConfiguration _exampleConfiguration;

}
```

Now you just need to build your module, and place its .jar file into the deploy folder of your bundle along with the built-in bundles.

If you created a custom configuration interface also, and you would like to pass the option values through a .cfg file, then you also have to create the .cfg file in the format as described in the first section of this document (e.g. `com.liferay.cloning.example.configuration.ExampleConfiguration-default.cfg`), and copy the .cfg file into osgi/configs.