How to Migrate from Liferay Portal 6.2 to Liferay Digital Experience Platform

LIFERAY

Table of Contents

Introduction

As customer demand for always-on and everywhere digital experiences has risen, so has the need for companies to equip themselves with the right technologies to deliver great digital experiences and remain agile for future digital innovations. Liferay Digital Experience Platform (DXP) enables digital businesses to manage and deliver customer experiences that are consistent and connected across digital touchpoints, including mobile, desktop, kiosk, smart devices and more.

A migration to Liferay DXP is an investment in addressing immediate needs in digital experience while laying the foundation to serve an increasingly connected digital audience. The migration places your business in the best position to take advantage of Liferay's latest developments for digital businesses including in mobile experience delivery, multichannel engagement, content targeting and more.

This whitepaper aims to set a framework for Liferay's recommended migration path for your organization. A major technology transition is an endeavor requiring a deep analysis of your business requirements, careful planning, testing and execution in order to be successful. Before you start on your planning and execution, Liferay's Global Services team, our group of professional consultants with experience in migrating customers to the Liferay DXP, can help you in a critical analysis of your needs through the *Liferay Upgrade Analysis Program*. Pairing this comprehensive analysis with an awareness of the key considerations needed in a migration to Liferay DXP, you can make the most informed decisions for your company to start benefiting from the Liferay platform.

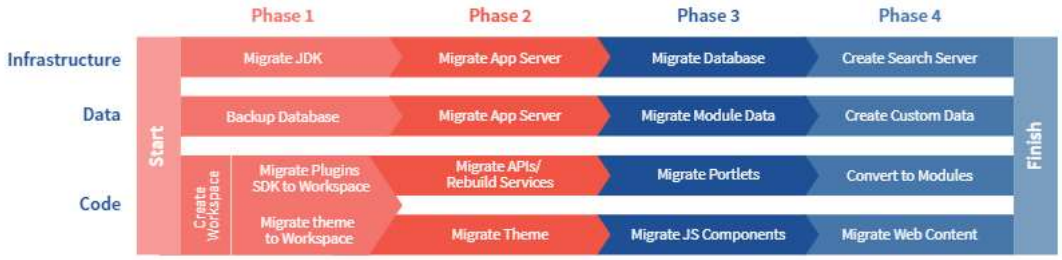
Setting the Right Timeline

Migrations require data modifications, code modifications and infrastructure changes. The process inherently involves a great deal of risk. Answering the following questions now can help you set a proper timeline and set expectations to better manage the risks.

1. **What version am I on?** If you are a few versions away from the current Liferay version, you should remember that your data will need to go through the full migration path of previous versions before you will eventually be migrated to Liferay DXP. This means that if you are on Liferay Portal 6.1, your data will first be updated to 6.2 before it is migrated to Liferay DXP. The upgrade tool can handle this for you, but you may want to take a more manual approach.
2. **How much data do I have?** If your project has a lot of data, it is essential to have a properly indexed database. Also set aside more time if you have a larger database.
3. **How much of my existing web content, templates and structures will I need to migrate?** Many people decide to write new content after a migration while others decide to reuse their current content.
4. **How many portlets will I need to migrate?** Not all portlets will need to be migrated for Liferay DXP. A proper analysis will give you a better estimate on how much time you will need and the number of engineers to devote for this process.
5. **Am I overriding many JSPs?** JSPs have changed a lot since versions 6.1 and 6.2. We recommend moving away from overriding JSPs completely, if possible. A number of JSPs have extensions you can plug into without resorting to this.

6. **Do I have an EXT to migrate?** The good news is that with Liferay DXP, we've created many more extension points, so the EXT can finally be retired.
7. **Am I planning to convert to OSGi bundles?** It will take more time, but the investment may well be worth it.

Sample Liferay DXP Upgrade Path



Infrastructure Changes

Compatibility Matrix

Now that you're ready to begin your migration, first start with making sure your environment is up to date with the *Liferay DXP compatibility matrix*. Many environments have gone end-of-life since 6.2 was released, even more if you are coming from an older version. It is essential that your system is running on a supported environment to receive proper support.

Search

The biggest change you will notice is that Liferay now requires the search engine to run as a separate JVM, either on the same or on a dedicated physical/virtual server.

We've found that search has grown to be a vital part of any website. The performance and scalability profile of a search engine is drastically different from that of a portal actively serving web impressions. In the past, Liferay Portal either supported an embedded Lucene search engine or a remotely deployed Solr search engine. With Liferay DXP, we will continue to offer Solr as a supported remote search engine. However, the preference is to use Elasticsearch. This allows the same engine to be used in embedded mode during development and in a standalone mode for production.

If you were previously using Solr, you will only need to ensure you are using the newest Solr module and upgrade to 5.0. Please refer to our search documentation to properly configure your search server. Also note that you can run your search server on the same server, but separate JVM, if you have limited resources, though this is not recommended.

JDK

Another change to the compatibility matrix is that we've moved to JDK8. All app servers are already JDK8 compatible on the compatibility matrix. Please ensure your app server is properly configured.

Deployment Plan

With the introduction of the OSGi container in Liferay DXP, your deployment plan will need to change as there are now a few different ways to deploy your plugins.

- **WARs** are traditional Liferay Plugin web apps (e.g., *-theme.war, *-portlet.war, *-web.war) to which we are all accustomed. WARs are not recommended in Liferay DXP because you would not benefit from the explicit dependency and lifecycle management models provided by OSGi.
- **Bundles/Modules** are plugins you've converted to an OSGi bundle. They are just simple Java JARs with OSGi metadata. Bundles can only be deployed into the OSGi container. Bundles cannot access services deployed as WARs besides Liferay's core services. This is the recommended approach for all new development and will be the approach Liferay takes for all new development.
- **WABs** are web archive bundles. If you deploy a WAR to the OSGi container, Liferay will convert the WAR into a WAB. This will give you all the benefits of a bundle without doing the conversion. This is the recommended approach for deploying legacy applications built for older versions of Liferay.

The Liferay auto deploy directory now deploys to the OSGi container by default. The only way to deploy to your web application's deploy folder is to do this through your application server's own deployment mechanisms. Any artifacts copied into Liferay's deploy folder will automatically be pulled into the Liferay OSGI container.

CAUTION: You should never deploy Liferay artifacts (WARs, modules, WABs) directly using your application server's deployment tools.

When deploying OSGi bundles in an application server's managed cluster (e.g., JBoss domain mode, WebLogic w/ Node Manager, WebSphere w/ Deployment Manager), you will need to rely upon Liferay's Cluster Deployment Helper.

This tool will take specified deployment artifacts and bundle them into a specialized WAR. When the application server deploys and starts the WAR, the startup mechanisms will place the OSGi modules into Liferay Digital Enterprise's OSGi deployment folder on the application server.

Database Upgrade

Before You Start

Now that your infrastructure has been migrated to supported versions, we can focus our attention on the data. The first thing we should do is ensure a proper backup is in place for us in case of failures.

Next ensure you are running permission algorithm 6 if you are coming from 6.1.

Also check that all your database indexes have been applied correctly. A missing index can cause a migration to really slow down. If you are running into a slow migration later on, you may want to come back and add additional temporary indexes to help speed them up.

We also recommend you disable search indexing during this process. To achieve that, you should add a file called *com.liferay.portal.search.configuration.IndexStatusManagerConfiguration.cfg* into your *osgi/configs/* folder with the following content:

```
indexReadOnly=true
```

By doing this, you will avoid indexing and save time during the migration process. Once you have migrated your Liferay site, make sure to set the property to false so that you can index all objects from control panel.

Upgrade Tool

In Liferay DXP, the database upgrades have been moved to a standalone tool.

To run the tool:

```
java -jar com.liferay.portal.tools.db.upgrade.client.jar
```

The process requires three files to be configured before it can run:

1. App-server.properties,
2. portal-upgrade-database.properties,
3. portal-upgrade-ext.properties.

You can do it manually or the tool can walk you through it. The data migration is now broken up into two parts. The core migration is similar to what you've seen in the past. The next part will update the OSGi modules. By default, the upgrade tool is configured to address both automatically.

Troubleshooting

If your migration ran successfully, you can skip this section. If you ran into issues, here are some tips to help you.

Start with the tips we presented before you began. Most migration issues are due to corrupted data. The only way to fix issues of this kind is to fix or remove the corrupt data. Unfortunately, this means that you will have to restart the migration after fixing the issue. This can quickly become tedious so the tips presented here are methods you can take to help prevent restarting from scratch. We will show you areas where you can safely take snapshots of your database and set that as the new reset point. Remember that you should never replace these reset points; label them carefully so you have a history.

One of the advances for Liferay DXP is separation between a logical "core" and a series of "modules." This allows us to take smaller incremental steps to help triage and resolve data related issues.

Isolated Core Upgrade

NOTE: Isolating and migrating the site core and the modules should only be done as part of debugging, building data cleanup scripts and testing in a non-production environment. We generally advise testing the migration against a copy of your production database and resolving all issues before attempting to perform a full migration in production.

You can configure the site to only migrate the core and not the modules, by adding a file called *com.liferay.portal.upgrade.internal.configuration.ReleaseManagerConfiguration.cfg* in the *osgi/configs/* folder with the following content:

```
autoUpgrade=false
```

The core migration is split into versions. You can configure which ones to run using *portal-ext.properties.*

```
upgrade.processes.master=\
    com.liferay.portal.upgrade.UpgradeProcess_6_0_12_to_6_1_0\,\
    com.liferay.portal.upgrade.UpgradeProcess_6_1_1\,\
    com.liferay.portal.upgrade.UpgradeProcess_6_2_0\,\
    com.liferay.portal.upgrade.UpgradeProcess_7_0_0\,\
    com.liferay.portal.upgrade.UpgradeProcess_7_0_1
```

You should also disable the VerifyProcess from executing by setting the portal.properties:

```
verify.frequency=0
```

You can reactivate the VerifyProcess after the migration has been successfully completed. If you are coming from a version prior to 6.2, you may try to configure the *upgrade.process.master* property for a more incremental test of the migration process. This will allow you to restore the database to the previous version. Of course, this should only be done to test your migration in a non-production database. Once you have successfully tested and fixed all data integrity issues, you should perform the migration as previously discussed.

If the issues occur while migrating the modules, you will need to configure your upgrade tool to manually execute the modules' migrations. Complete the core migration and take a snapshot. If you have issues, you can restart from there.

OSGi modules can be updated individually. These offer safe snapshots if they execute successfully.

For more advanced techniques, you can use the debugger. The advantage of this is that, in some cases, you can fix the corrupt data in memory and it will be saved in the database, fixing your data without a restart. This method requires a high level of background knowledge about the tables, and is only recommended for the most seasoned Liferay developers. This technique also opens up a new option during module migrations because OSGi bundles are updated in steps. You can use a debugger to stop at any step and take a snapshot of your database there.

Migrating the Modules

To run the migrations for the modules, you can use the Gogo shell.

1. Connect to the shell by executing *telnet localhost 11311*.
2. Use the available commands in the upgrade namespace. For example:
    a. upgrade:list
    b. upgrade:execute
    c. upgrade:check
    d. verify:list
    e. verify:execute

By typing *upgrade:list*, the console will show you the modules you can migrate since all their migration dependencies are covered.

If you do not see any modules, that is because we need to migrate their dependencies first. You could enter the command *scr:info {upgrade_qualified_class_name}* to check which dependencies are unsatisfied. For example:

- *scr:info com.liferay.journal.upgrade.JournalServiceUpgrade*

By typing *upgrade:list {module_name}*, the console will show you the steps you have to complete for migrating your module. To understand how this works, it can be useful to see an example.

If you execute that command for the bookmarks service module, you will get this:

```
Registered upgrade processes for com.liferay.bookmarks.service 1.0.0
      {fromSchemaVersionString=0.0.1, toSchemaVersionString=1.0.0-step-3,
upgradeStep=com.liferay.bookmarks.upgrade.v1_0_0.UpgradePortletId@497d1106}
      {fromSchemaVersionString=1.0.0-step-1, toSchemaVersionString=1.0.0,
upgradeStep=com.liferay.bookmarks.upgrade.v1_0_0.UpgradePortletSettings@31e8c69b}
      {fromSchemaVersionString=1.0.0-step-2, toSchemaVersionString=1.0.0-step-1,
upgradeStep=com.liferay.bookmarks.upgrade.v1_0_0.UpgradeLastPublishDate@294703b6}
      {fromSchemaVersionString=1.0.0-step-3, toSchemaVersionString=1.0.0-step-2,
upgradeStep=com.liferay.bookmarks.upgrade.v1_0_0.UpgradeClassNames@7544b6e5}
```

This means that there is an available process to upgrade bookmarks from 0.0.1 version to 1.0.0. To complete it, you would need to execute four steps and the first one is the one that starts on the initial version and finishes in the first step of the target version (the highest step number, step-3 for this example), *UpgradePortletId* in this case. The latest step is the one that starts in the latest step of the target version (the lowest step number, step-1) and finishes in the target version (1.0.0), *UpgradePortletSettings* in this case.

By typing *upgrade:execute {module_name}*, you will migrate a module. It is important to take into account that, if there is an error during the process, you will be able to restart the process from the latest executed step successfully instead of executing the whole process again. You could check the status of your migration by executing *upgrade:list {module_name}*.

By typing *upgrade:check* at the Gogo shell, it will show you the modules that have not reached the final version. Thus you will have a way to identify the modules whose upgrades have failed at the end of the process.

To understand how this command works, please consider this example: Picture that the upgrade for module *com.liferay.dynamic.data.mapping.service* fails in the step *1.0.0-step-2*. If you execute the command upgrade:check at this moment you will get:

```
Would upgrade com.liferay.dynamic.data.mapping.service from 1.0.0-step-2 to 1.0.0
and its dependent modules
```

That means that you will need to fix the issue and execute the migration for that module again. Notice that dependent modules for *com.liferay.dynamic.data.mapping.service* need to be migrated once the first one is migrated properly.

Also, you can execute the verify process from command line using *verify:list*. This checks all available verify processes. Execute *verify:execute {verify_qualified_name}* to run it.

Migrating Your Code

Finally, we are covering how to migrate your code to Liferay DXP. If you have multiple members in your team, this part can be started in tandem with the data migration.

Breaking Changes

One of the most difficult parts of migrating your codebase is knowing what's changed. For Liferay DXP, we've documented the changes, which can be found at *this link*. It presents a chronological list of changes that break existing functionality, APIs or contracts with third-party Liferay developers or users. Some of the types of changes documented in the file include:

- Functionality that is removed or replaced.
- API incompatibilities - Changes to public Java or JavaScript APIs.
- Changes to context variables available to templates.
- Changes in CSS classes available to Liferay themes and portlets.
- Configuration changes - Changes in configuration files, like *portal.properties*, *system.properties*, etc.
- Execution requirements - Java version, J2EE Version, browser versions, etc.
- Deprecations or end of support - For example, warning that a certain feature or API will be dropped in an upcoming version.
- Recommendations - For example, recommending using a newly introduced API that replaces an old API, in spite of the old API being kept in Liferay for backwards compatibility.

It is important that you familiarize yourself with the document to understand what changes you may have to make with your codebase. We recommend reading the document to get a general feel of how Liferay DXP will continue evolve in future versions.

To view the current breaking changes for Liferay DXP, *visit the list here*.

Liferay Command Line Tools

As part of DXP, Liferay invested heavily on improving the developer experience. One of the improvements is the move to using Gradle as the primary build infrastructure.

Liferay Workspace

In Liferay DXP, we recommend that you move to our new project structure called Liferay Workspace. It is built on Gradle and provides support for your Plugins SDK based projects. It will also leverage all the new theme tooling we've built in Liferay DXP and integrates with Liferay Developer Studio.

Your new project will look like this:

- Project-folder
  - Modules
  - Plugins-sdk
  - Themes
- If you are coming from a project that leveraged the Plugins SDK, you will be able to move your Plugins SDK as one of the folders within Liferay Workspace. Keep in mind, the Plugins SDK only supports creating WARs. You will not be able to create OSGi bundles from the Plugins SDK. Theme support has also been deprecated within the Plugins SDK.

Liferay Blade

Alongside Workspace, we have a new command line tool called Liferay Blade. This tool allows you to create applications, extensions, etc. as you could in a Plugins SDK, but it also provides additional functionality. It can start your Liferay server or automatically deploy your project as you make changes. It is also the way you create a new Liferay Workspace.

Installation

OS X OR LINUX

```
curl https://raw.githubusercontent.com/liferay/liferay-blade-cli/master/
installers/global | sudo sh
```

WINDOWS

First install JPM. Visit the JPM4J [Windows installation] setup guide: *https://www.jpm4j.org/#!/md/windows*.

```
jpm install com.liferay.blade.cli.jar
```

Usage

```
blade init folderName
cd folderName
```

Unzip a new Liferay 7.0 Plugins SDK into your new Workspace and name it plugins-sdk. You can now copy over your existing plugins into the new sdk.

Migrating a 6.2 WAR to a Liferay DXP-supported WAR

The first step you must take is migrating your 6.2 portlet to a Liferay DXP application. Even if you plan on converting your portlet to OSGi modules, we recommend that you update your legacy WAR to be supported by Liferay DXP first. If you jump from a legacy 6.2 WAR to DXP modules, it will be much more difficult to debug and figure out which issues are related to API changes and which are related to the migration process.

Once you've copied your portlet into a Liferay 7.0 Plugins SDK instance, use the Upgrade Assistant mentioned above to find the breaking changes in your portlet and update them accordingly. Make sure to also update any Liferay dependencies you've specified to Liferay DXP (e.g., *ivy.xml, liferay-plugin-package.properties*, etc.).

When you've completed the migration process and have a Liferay DXP-supported WAR, you'll need to make the decision on whether you should convert your portlet to an OSGi module. We've outlined scenarios below that will help you make your decision.

Converting a Portlet to an OSGi Module

Now that we've created our workspace and updated your portlet to a Liferay DXP-supported WAR, we can consider if you want to convert your portlet to an OSGi module.

You should convert when:

- You have a very large application with many lines of code. For example, if there are many developers that are collaborating on an application concurrently and making changes frequently, separating the code into modules will increase productivity and provide the agility to release more frequently.
- Your plugin has reusable parts that you'd like to consume from elsewhere. For instance, suppose you have business logic that you're reusing in multiple different projects. Instead of copying that code into several different WARs and deploying those WARs to different customers, you can convert your application to modules and consume the services provided by those modules from other modules.

You should NOT convert when:

- You have a portlet that's JSR-168/286 compatible and you still want to be able to deploy it into another portlet container. If you want to retain that compatibility, it is recommended to stay with the traditional WAR model.
- You're using a complex legacy web framework that is heavily tied to the Java EE programming model, and the amount of work necessary to make that work with OSGi is more than you feel is necessary or warranted.
- Your plugin interacts with other JEE app server features, for instance EJBs, message driven beans, etc. Module-based applications are not as portable when they directly interact with the app server.
- Your legacy application's original intent is to have a limited lifetime.

If you decided to convert your portlet to an OSGi module, we'll walk you through it here.

For a portlet.

```
blade create [APPLICATION_NAME]
```

For a portlet with service builder:

```
blade create -t servicebuilder -p [ROOT_PACKAGE] [APPLICATION_NAME]
```

The first thing you will notice is that projects now use the standard maven project structure.

- plugin-name
  - src
    - main
      - java
      - resources
        - content
        - META-INF
          - resources
            - css
              - *.scss
            - *.jsp
  - bnd.bnd
  - build.gradle

In your workspace, the bnd.bnd file is very important, as it will automatically apply the *liferay-gradle-plugin* to your Gradle project. The *liferay-gradle-plugin* will apply the Gradle Java plugin along with other Liferay plugins that are very useful such as *css-builder*, *source-formatter* and *lang-builder.*

You will notice that you do not need a portlet.xml/liferay-portlet.xml file. The contents of that file should be moved into the portlet Java class.

```
@Component(
  immediate = true,
  property = {
    “com.liferay.portlet.display-category=category.sample”,
    “com.liferay.portlet.icon=/icon.png”,
    “javax.portlet.name=1”,
    “javax.portlet.display-name=Tasks Portlet”,
    “javax.portlet.security-role-ref=administrator,guest,power-user”,
    “javax.portlet.init-param.clear-request-parameters=true”,
    “javax.portlet.init-param.view-template=/view.jsp”,
    “javax.portlet.expiration-cache=0”,
    “javax.portlet.supports.mime-type=text/html”,
    “javax.portlet.resource-bundle=content.Language”,
    “javax.portlet.info.title=Tasks Portlet”,
    “javax.portlet.info.short-title=Tasks”,
    “javax.portlet.info.keywords=Tasks”,
  },
  service = Portlet.class
)
public class TasksPortlet extends MVCPortlet {
```

If you've created a service builder template, you will notice three different plugins generated for you.

plugin-name-api - This is where the interfaces of your services will live.
plugin-name-service - This is where the implementations of your services will live.
plugin-name-web - This is where your portlet will live.

This will encourage module development within your plugins also.

Migrating Themes

In Liferay DXP, we've created a new set of theme tools that frontend developers should be much more familiar with. They are built using Node.js, yo and gulp. If you have an existing theme in your Plugins SDK, you can migrate them to your new Liferay Workspace.

To migrate a single theme:

```
blade migrateTheme [THEME_NAME]
```

To migrate all themes use:

```
blade migrateTheme -a
```

Your themes will be moved to the new theme folder. Next we will need to convert the CSS styles from Bootstrap 2 to Bootstrap 3. We will need to leverage another npm package created by Liferay.

The next thing you will want to do is rename all your SASS files from *.css to *.sass. In Liferay DXP, SASS files use the proper extension. The SASS compilers will only compile files with the.scss file extension.

Install it by running:

```
npm install -g convert-bootstrap-2-to-3
```

Then you can update the file by running:

```
bs3 path/to/file
```

You can run this against HTML files, CSS files, and SASS files. This tool won't fix everything for you, but it will give you a great headstart. The rest is up to you.

Additional Resources

Upgrade to 6.2

Migrate to Liferay DXP

Creating a Liferay Workspace

Theme Tasks Documentation

Bootstrap Convert Documentation

Summary

For those who choose to go forward, a migration to Liferay Digital Experience Platform has the potential to unlock the full range of benefits in the latest version of the Liferay platform. Each company must determine for itself whether a migration will be beneficial for the direction of the business after carefully weighing the costs, risks, time frame, labor and business benefits involved.

Moving Forward

Our *Liferay Global Services* team is ready to help you form a personalized migration plan and offer you inside knowledge on setting your company up for success with Liferay. Learn more about Liferay Digital Experience Platform and the consulting services available to you by contacting *sales@liferay.com*.

Liferay makes software that helps companies create digital experiences on web, mobile, and connected devices. Our platform is open source, which makes it more reliable, innovative and secure. We try to leave a positive mark on the world through business and technology. Companies such as Adidas, Carrefour, Cisco Systems, Danone, Fujitsu, Lufthansa Flight Training, Siemens, Société Générale and the United Nations use Liferay. Visit us at www.liferay.com.

170405